Gilles Dowek

Jean-Pierre Archambault, Emmanuel Baccelli, Claudio Cimelli, Albert Cohen, Christine Eisenbeis, Thierry Viéville et Benjamin Wack

Préface de Gérard Berry, professeur au Collège de France

Informatique et sciences du numérique

Spécialité ISN en terminale S

Avec des exercices corrigés et des idées de projets

EYROLLES

Gilles Dowek est chercheur Inria, ses travaux portent sur les liens entre le calcul et le raisonnement. Il est lauréat du Grand prix de philosophie de l'Académie française pour son livre *Les Métamorphoses du Calcul.*

Jean-Pierre Archambault est professeur agrégé de mathématiques et président de l'association Enseignement public et informatique (EPI). **Claudio Cimelli** est inspecteur d'académie, inspecteur pédagogique régional en Sciences et techniques industrielles (STI) et conseiller TICE (technologies de l'information et de la communication pour l'enseignement) du recteur de Créteil. **Benjamin Wack** est docteur en informatique et professeur agrégé de mathématiques. **Emmanuel Baccelli, Albert Cohen, Christine Eisenbeis** et **Thierry Viéville** sont docteurs en informatique et chercheurs Inria. Leurs travaux respectifs portent sur les réseaux, la construction de programmes effectuant des milliers de calculs en parallèle, les limites physiques du calcul et la simulation du cerveau.

Suffit-il d'apprendre à utiliser les appareils dont on est entouré ? N'a-t-on pas envie d'en comprendre le fonctionnement ?

Ne préfère-t-on pas se former au cœur conceptuel et technique de l'informatique, plutôt que de rester les consommateurs serviles de ce qui se crée ailleurs ?

L'objectif de ce cours est d'introduire les quatre concepts de machine, d'information, d'algorithme et de langage qui sont au cœur de l'informatique, et de montrer comment ils fonctionnent ensemble. Dans la première partie, nous apprendrons à **écrire des programmes**, en découvrant les ingrédients dont les programmes sont constitués : l'affectation, la séquence et le test, les boucles, les types, les fonctions et les fonctions récursives. Dans la deuxième partie, nous apprendrons à **représenter les informations** que l'on veut communiquer, les stocker et les transformer – textes, nombres, images et sons. Nous apprendrons également à structurer et compresser de grandes quantités d'informations et à les protéger par le chiffrement. Dans la troisième partie, nous verrons que derrière les informations, il y a toujours des objets matériels : **ordinateurs**, **réseaux**, **robots**, etc. Nous découvrons comment fonctionnent toutes ces machines que nous utilisons tous les jours. Dans la quatrième partie, nous apprendrons quelques-uns des savoir-faire les plus utiles au XXIe siècle : ajouter des nombres exprimés en base deux, dessiner, retrouver une information par dichotomie, trier des informations et parcourir des graphes.

Ce cours comporte des chapitres élémentaires et avancés. Chacun contient une partie de cours, des sections de savoir-faire qui permettent d'acquérir les capacités essentielles, et des exercices, notés difficiles pour certains, avec corrigés lorsque nécessaire. Avec des compléments disponibles en ligne.

À qui s'adresse ce livre ?

Ce manuel de cours destiné aux élèves de terminale S ayant choisi la spécialité Informatique et sciences du numérique sera également lu avec profit par tous les professionnels de l'informatique, qu'ils soient autodidactes ou non, et tous ceux qui veulent apprendre l'informatique.

Au sommaire

LANGAGES • **Les ingrédients des programmes** • Modifier, comprendre, écrire et tester un programme • Instructions et expressions • Opérations • Accolades • Indenter un programme • **Boucles** • Boucles for et while • Imbriquer deux boucles • Non-terminaison • Commenter un programme • **Types** • Types de base • Déclaration, portée et initialisation des variables • Tableaux • Chaînes de caractères • **Les fonctions** • Isoler une instruction • Passer des arguments • Récupérer une valeur • **La récursivité** • Fonctions qui appellent des fonctions • Fonctions qui s'appellent elles-mêmes • **La notion de langage formel** • Grammaire et sémantique • **REPRÉSENTER L'INFORMATION** • **Nombres entiers et à virgule** • Compter en base n • **Caractères et textes** • ASCII binaire • Écrire une page en HTML • **Images et sons** • Numériser une image • Notion de format • Tailles de fichier • **Fonctions booléennes** • Fonctions non, et, ou • **Structurer l'information** • Persistance des données • Notion de fichier • Organiser des fichiers en une arborescence • Liens et hypertextes • Revers de l'hypermnésie • **Compresser, corriger, chiffrer** • **MACHINES** • **Portes booléennes** • **Temps et mémoire** • **Organisation d'un ordinateur** • **Réseaux** • Protocoles • Couches • Trouver les adresses MAC et IP • Déterminer le chemin suivi par l'information • Régulation du réseau global • **Robots** • Composants d'un robot • Programmer un robot • **ALGORITHMES** • **Ajouter deux nombres exprimés en base deux** • **Dessiner** • Formats d'images • Transformer les images • **Dichotomie** • Recherche en table • Conversion analogique-numérique • Trouver le zéro d'une fonction • **Trier** • Tri par sélection et par fusion • Efficacité des algorithmes • **Parcourir un graphe** • États et transitions • **Idées de projets.**

Code article : G13543
ISBN : 978-2-212-13543-5

Inria
INVENTEURS DU MONDE NUMÉRIQUE

Ouvrage publié avec le concours de l'EPI, la SIF et Inria 

www.editions-eyrolles.com

Informatique et sciences du numérique

Spécialité ISN en terminale S

Avec des exercices corrigés et idées de projets

Dans la collection Noire

P. Cegielski. – **Conception de systèmes d'exploitation**. *Le cas Linux*. N°G11479, 2e édition, 2004, 680 pages.

J. Engels. **HTML5 et CSS3**. *Cours et exercices corrigés*. N°13400, 2012, 550 pages.

G. Swinnen. **Apprendre à programmer avec Python 3**. N°13434, 3e édition, 2012, 435 pages.

H. Bersini. **La programmation orientée objet**. *Cours et exercices en UML 2 avec Java 6, C# 4, C++, Python, PHP 5 et LinQ*. N°12806, 5e édition, 2011, 644 pages.

É. Sarrion. – **jQuery et jQuery UI**. N°12892, 2011, 132 pages.

A. Brillant. **XML - Cours et exercices**. N°12691, 2e édition, 2010, 336 pages.

Chez le même éditeur

Créer son site web avec un CMS

F.-X. et L. Bois. – **WordPress 3 pour le blogueur efficace.** N°12829, 2011, 358 pages.

H. Cocriamont. **Réussir son premier site Joomla! 2.5**. N°13425, 2012, 250 pages environ.

T. Parisot. – **Réussir son blog professionnel.** N°12768, 2e édition, 2010, 322 pages.

V. Isaksen, T. Tardif. – **Joomla 2.5 et Virtuemart 2.** *Réussir sa boutique en ligne*. N°12804, 3e édition à paraître, 2012, 350 pages environ.

Y. Brault, préface d'Edwy Plenel. – **Concevoir et déployer ses sites web avec Drupal 6 et 7.** N°12780, 2e édition, 2010, 420 pages.

Développer soi-même son site web avec HTML, CSS, PHP, JavaScript

R. Rimelé. – **HTML5.** N°12982, 2011, 600 pages.

F. Draillard. – **Premiers pas en CSS et HTML**. N°13338, 2011, 464 pages.

R. Goetter. – **CSS avancées.** *Vers HTML5 et CSS3*. N°13405, 2e édition, 2012, 385 pages.

R. Goetter. – **CSS 2 : pratique du design web**. N°12461, 3e édition, 2009, 340 pages.

R. Rimelé. – **Mémento HTML5.** N°13420, 2012, 14 pages.

R. Goetter. – **Mémento CSS 3**. N°13281, 2011, 14 pages.

É. Daspet et C. Pierre De Geyer. **PHP 5 avancé**. N°13435, 6e édition, 2012, 870 pages.

C. Porteneuve. – **Bien développer pour le Web 2.0**. N°12391, 2e édition, 2008, 674 pages.

Développer pour le Web mobile

F. Daoust, D. Hazaël-Massieux. – **Bonnes pratiques pour le Web mobile.** *Conception et développement*. N°12828, 2011, 300 pages.

T. Baillet. **Créer son thème WordPress mobile** *en HTML5 et CSS3*. N°13441, 2012, 128 pages.

É. Sarrion. – **XHTML/CSS et JavaScript pour le Web mobile**. *Développement iPhone et Android avec et iUI et XUI*. N°12775, 2010, 274 pages.

É. Sarrion. – **jQuery Mobile**. N°13388, 2012, 601 pages.

Ressources autour du Web : design, ergonomie, bonnes pratiques

A. Boucher. – **Ergonomie web.** *Pour des sites web efficaces*. N°13215, 3e édition, 2011, 380 pages.

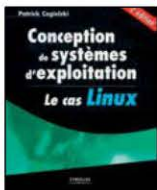
A. Boucher. – **Ergonomie web illustrée**. *60 sites à la loupe*. N°12695, 2010, 302 pages (Design & Interface).

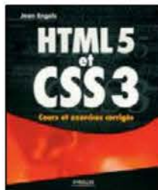
A. Boucher. – **Mémento Ergonomie web.** N°12698, 2e édition, 2010, 14 pages.

E. Sloïm. – **Mémento Sites web.** *Les bonnes pratiques*. N°12802, 3e édition, 2010, 18 pages.

O. Andrieu. – **Réussir son référencement web.** N°13396, 2012, 480 pages.

I. Canivet. – **Bien rédiger pour le Web.** *Stratégie de contenu pour améliorer son référencement*. N°12883, 2e édition, 2011, 540 pages.

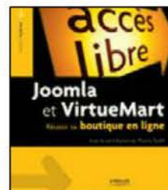
















Gilles Dowek

Jean-Pierre Archambault, Emmanuel Baccelli, Claudio Cimelli,
Albert Cohen, Christine Eisenbeis, Thierry Viéville et Benjamin Wack
Préface de Gérard Berry, professeur au Collège de France

Informatique et sciences du numérique

Spécialité ISN en terminale S

*Avec des exercices corrigés
et idées de projets*

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

**Ouvrage publié avec le concours
de l'association EPI – Enseignement Public et Informatique,
de la SIF – Société Informatique de France,
et de l'Institut public de recherche en sciences du numérique – Inria.**

***Remerciements à Anne Bougnoux (relecture) et Gaël Thomas (maquette),
ainsi qu'à Raphaël Hertzog, Pierre Néron, Christine Paulin, Grégoire Péan, Jonathan Protzenko
et Dominique Quatravaux pour leurs témoignages.***

***Merci à Randall Munroe du site XKCD pour les dessins d'ouverture de partie adaptés de l'anglais
ainsi qu'à Rémi Cieplicki de www.DansTonChat.com pour nous avoir autorisés à utiliser leur logo.***

***Illustrations de Camille Vorng (cactus, boîtes, arborescences),
Laurène Gibaud et Bernard Sullerot (circuits logiques, opérations binaires, schémas, labyrinthes)***

Photographies d'ouvertures de chapitres
*Alan Turing (aimable autorisation de la Sherborne School, merci à Rachel Hassall),
John Backus (PIerre.Lescanne, CC-BY-SA-3.0), Grace Hopper (James S. Davis, domaine public),
Gilles Kahn (marcstephanegoldberg – Flickr), Gordon Plotkin (merci à lui d'avoir accepté de nous fournir une photographie),
John McCarthy (null0 – Flickr, CC BY 2.0), Robin Milner (http://www.cl.cam.ac.uk/archive/rm135/),
Dana Scott (Andrej Bauer –http://andrej.com/mathematicians), Claude Shannon (Tekniska museet – Flickr, CC BY-SA 2.0),
Tim Berners-Lee (Paul Clarke, CC-BY-2.0), Ronald Rivest (carback1, CC BY 2.0),
Adi Shamir (Ira Abramov de Even Yehuda, Israel, CC-BY-SA-2.0), Len Adleman (len adlmen, CC-BY-SA-3.0),
Frances Allen (Rama, CC-BY-SA-2.0-fr), John Von Neumann (LANL, domaine public),
Vinton Cerf et Robert Kahn (Paul Morse, domaine public), Ada Lovelace (Alfred Edward Chalon, domaine public),
Ivan Sutherland (Dick Lyon, CC-BY-SA-3.0), Donald Knuth (Jacob Appelbaum, CC-BY-SA-2.5),
Philippe Flajolet (Luc Devroye, CC-BY-SA-3.0), Joseph Sifakis (Rama, CC-BY-SA-2.0-fr),
Christopher Strachey (http://www.rutherfordjournal.org/article040101.html), Gottlob Frege (inconnu, domaine public),
Muhammad al-Khwarizmi et Samuel Morse (inconnu, domaine public),
Thomas Flowers (http://www.ithistory.org/honor_roll/fame-detail.php?recordID=444 – merci à l'équipe de IT History pour leur aimable autorisation), Otto Schmitt (http://160.94.102.47/index.htm), Norbert Wiener (Konrad Jacobs, CC-BY-SA-2.0-de)*

Autres images
Qui est-ce *est un jeu développé par la société Theora Design (http://theoradesign.com)
et distribué en France par MB (Idées de projets)*
La Joconde*, tableau de Léonard de Vinci (chapitre 19) et* L'Annonciation*, tableau de Sandro Botticelli (chapitre 19)*
Robolab : par Mirko Tobias Schäfer (chapitre 17)
Thyroïdectomie assistée par un robot : CHU de Nîmes (http://www.chu-nimes.fr/espace-presse-galerie-photos.html) (chapitre 17)
Robot mOway : http://www.moway-robot.com, http://www.adrirobot.it/moway/moway_circuito.htm (chapitre 17)

Préface

L'année 2012 voit l'entrée de l'informatique en tant qu'enseignement de spécialité en classe de Terminale scientifique. Cette entrée devenait urgente, car l'informatique est désormais partout. Créée dans les années 1950 grâce à une collaboration entre électroniciens, mathématiciens et logiciens (ces derniers en ayant posé les bases dès 1935), elle n'a cessé d'accélérer son mouvement depuis, envahissant successivement l'industrie, les télécommunications, les transports, le commerce, l'administration, la diffusion des connaissances, les loisirs, et maintenant les sciences, la médecine et l'art, tout cela en créant de nouvelles formes de communication et de relations sociales. Les objets informatiques sont maintenant par milliards et de toutes tailles, allant du giga-ordinateur équipé de centaines de milliers de processeurs aux micro-puces des cartes bancaires ou des prothèses cardiaques et auditives, en passant par les PC, les tablettes et smartphones, les appareils photos, ou encore les ordinateurs qui conduisent et contrôlent les trains, les avions et bientôt les voitures. Tous fonctionnent grâce à la conjonction de puces électroniques et de logiciels, objets immatériels qui décrivent très précisément ce que vont faire ces appareils électroniques. Au XXIe siècle, la maîtrise du traitement de l'information est devenue aussi importante que celle de l'énergie dans les siècles précédents, et l'informatique au sens large est devenue un des plus grands bassins d'emploi à travers le monde. Cela implique que de nombreux lycéens actuels participeront à son essor dans l'avenir.

Ces jeunes lycéens sont bien sûr très familiers avec les appareils informatisés. Mais ce n'est pas pour cela qu'ils en comprennent le fonctionnement, même

sur des plans élémentaires pour certains. Une opinion encore fort répandue est qu'il n'y a pas besoin de comprendre ce fonctionnement, et qu'il suffit d'apprendre l'usage des appareils et logiciels. À l'analyse, cette opinion apparemment naturelle s'avère tout à fait simpliste, avec des conséquences néfastes qu'il faut étudier de près. Pour faire un parallèle avec une autre discipline, on enseigne la physique car elle est indispensable à la compréhension de la nature de façon générale, et aussi de façon plus spécifique au travail de tout ingénieur et de tout scientifique, c'est-à-dire aux débouchés naturels de beaucoup d'élèves de terminale scientifique. Mais qui penserait qu'il suffit de passer le permis de conduire pour comprendre la physique d'un moteur ou la mécanique une voiture ? Or, nous sommes tous autant confrontés à l'informatique qu'à la physique, même si elle ne représente pas un phénomène naturel préexistant ; comme pour la physique, les ingénieurs et scientifiques devront y être au moins autant créateurs que consommateurs. Pour être plus précis, sous peine de ne rester que des consommateurs serviles de ce qui se crée ailleurs, il est indispensable pour notre futur de former au cœur conceptuel et technique de l'informatique tout élève dont le travail technique sera relié à l'utilisation avancée ou à la création de l'informatique du présent ou du futur. Il est donc bien naturel que la nouvelle formation à l'informatique s'inaugure en terminale scientifique. Mais elle devra immanquablement ensuite être élargie à d'autres classes, car tout élève sera concerné en tant que futur citoyen.

Pour être efficace, toute formation scolaire demande un support adéquat. Ce premier livre va jouer ce rôle pour l'informatique, en présentant de façon pédagogique les quatre composantes scientifiques et techniques centrales de son cœur scientifique et technique : langages de programmation, numérisation de l'information, machines et réseaux, et algorithmes. Il a été écrit par des chercheurs et enseignants confirmés, tous profondément intéressés par le fait que les élèves comprennent, assimilent et apprécient les concepts et techniques présentées. Il insiste bien sur deux points essentiels : le fait que ces quatre composantes sont tout à fait génériques, c'est-à-dire valables pour tous les types d'applications, des méga-calculs nécessaires pour étudier l'évolution du climat aux calculs légers et rapides à effectuer dans les micro-puces enfouies partout, et le fait que les concepts associés resteront valables dans le temps. En effet, si les applications de l'informatique évoluent très vite, son cœur conceptuel reste très stable, au moins au niveau approprié pour la terminale scientifique. L'enseigner de façon adéquate est nécessaire autant à la compréhension des bases qu'à tout approfondissement ultérieur. À n'en pas douter, cet ouvrage y contribuera.

Gérard Berry, directeur de recherche Inria
Professeur au Collège de France,
Membre de l'Académie des sciences, de l'Académie des technologies,
et de l'Academia Europaea

Table des matières

Avant-propos

Il y a un siècle, il n'y avait pas d'ordinateurs ; aujourd'hui, il y en a plusieurs milliards. Ces ordinateurs et autres *machines* numériques que sont les réseaux, les téléphones, les télévisions, les baladeurs, les appareils photos, les robots, etc. ont changé la manière dont nous :

- concevons et fabriquons des objets,
- échangeons des informations entre personnes,
- gardons trace de notre passé,
- accédons à la connaissance,
- faisons de la science,
- créons et diffusons des œuvres d'art,
- organisons les entreprises,
- administrons les états,
- etc.

Si les ordinateurs ont tout transformé, c'est parce qu'ils sont polyvalents, ils permettent de traiter des *informations* de manières très diverses. C'est en effet le même objet qui permet d'utiliser des logiciels de conception assistée par ordinateur, des machines à commande numérique, des logiciels de modélisation et de simulation, des encyclopédies, des cours en ligne, des bases de données, des blogs, des forums, des logiciels de courrier électronique et de messagerie instantanée, des logiciels d'échange de fichiers, des logiciels de lecture de vidéos et musique, des tables de mixage numériques, des archives numériques, etc.

Cette polyvalence s'illustre aussi par le nombre d'outils que les ordinateurs ont remplacé : machines à écrire, téléphones, machines à calculer, télévisions, appareils photos, électrophones, métiers à tisser…

En fait, les ordinateurs sont non seulement capables de traiter des informations de manières diverses, mais également de toutes les manières possibles. Ce sont des machines universelles.



En 1936, soit quelques années avant la construction des premiers ordinateurs, **Alan Turing** (1912-1954) – et en même temps que lui Alonzo Church – a étudié les liens entre les notions d'algorithme et de raisonnement mathématique.

Cela l'a mené à imaginer un procédé de calcul, les machines de Turing, et à suggérer que ce procédé de calcul puisse être universel, c'est-à-dire capable d'exécuter tous les algorithmes possibles.

Un procédé systématique qui permet de traiter des informations s'appelle un *algorithme*. Ainsi, on peut parler d'algorithmes de recherche d'un mot dans un dictionnaire, d'algorithmes de chiffrement et de déchiffrement, d'algorithmes pour faire des additions et des multiplications, etc. De manière plus générale, un algorithme est un procédé systématique qui permet de faire quelque chose. Par exemple une recette de cuisine est un algorithme.

Traiter des informations

Traiter des informations signifie appliquer, d'une manière systématique, des opérations à des symboles. La recherche d'un mot dans un dictionnaire, le chiffrement et le déchiffrement d'un message secret, l'addition et la multiplication de deux nombres, la fabrication des emplois du temps des élèves d'un lycée ou des pilotes d'une compagnie aérienne, le calcul de l'aire d'une parcelle agricole ou encore le compte des points des levées d'un joueur au Tarot sont des exemples de traitements d'informations.

La notion d'algorithme est très ancienne. Depuis la nuit des temps, les hommes ont conçu et appris des algorithmes, pour fabriquer des objets en céramique, tisser des étoffes, nouer des cordages ou, simplement, préparer des aliments.

Le bouleversement survenu au milieu du XXe siècle tient à ce que les hommes ont cessé d'utiliser exclusivement ces algorithmes à la main ; ils ont commencé à les faire exécuter par des machines, les ordinateurs. Pour y parvenir, il a fallu exprimer ces algorithmes dans des *langages* de programmation, accessibles aux ordinateurs. Ces langages sont différents des langues humaines en ce qu'ils permettent la communication non pas entre êtres humains, mais entre les êtres humains et les machines.

ALLER PLUS LOIN Des algorithmes aussi vieux que l'écriture

Il y a quatre mille ans, les scribes et les arpenteurs, en Mésopotamie et en Égypte, mettaient déjà en œuvre des algorithmes pour effectuer des opérations comptables et des calculs d'aires de parcelles agricoles. La conception d'algorithmes de traitement de l'information semble remonter aux origines mêmes de l'écriture. Dès l'apparition des premiers signes écrits, les hommes ont imaginé des algorithmes pour les transformer.

L'informatique est donc née de la rencontre de quatre concepts très anciens :

- machine,
- information,
- algorithme,
- langage.

Ces concepts existaient tous avant la naissance de l'informatique, mais l'informatique les a profondément renouvelés et articulés en une science cohérente.

Structure de l'ouvrage

L'objectif de ce cours est d'introduire les quatre concepts de machine, d'information, d'algorithme et de langage, mais surtout de montrer la manière dont ils fonctionnent ensemble. Quand nous étudierons les algorithmes fondamentaux, nous les exprimerons souvent dans un langage de programmation. Quand nous étudierons l'organisation des machines, nous verrons comment elles permettent d'exécuter des programmes exprimés dans un langage de programmation. Quand nous étudierons la notion d'information, nous verrons des algorithmes de compression, de chiffrement, etc.

Ce livre est donc organisé en quatre parties regroupant vingt-deux chapitres, dont certains d'un niveau plus avancé (indiqués par un astérisque) :

- Dans la **première partie « Langages »**, nous apprendrons à écrire des programmes. Pour cela, nous allons découvrir les ingrédients dont les programmes sont constitués : l'affectation, la séquence et le test (**chapitre 1**), les boucles (**chapitre 2**), les types (**chapitre 3**), les fonctions (**chapitre 4***) et les fonctions récursives (**chapitre 5***). Pour finir, nous nous pencherons sur la notion de langage formel (**chapitre 6***). Dès que l'on commence à maîtriser ces concepts, il devient possible de créer ses propres programmes.
- Dans la **deuxième partie, « Informations »**, nous abordons l'une des problématiques centrales de l'informatique : représenter les informations que l'on veut communiquer, stocker et transformer. Nous apprendrons à représenter les nombres entiers et les nombres à virgule (**chapitre 7**), les caractères et les textes (**chapitre 8**), les images et les sons (**chapitre 9**). La notion de valeur booléenne, ou de bit, qui apparaît dans ces trois chapitres, nous mènera naturellement à la notion de fonction booléenne (**chapitre 10**). Nous apprendrons ensuite à structurer de grandes quantités d'informations (**chapitre 11***), à optimiser la place occupée grâce à la compression, corriger les erreurs qui peuvent se produire au moment de la transmission et du stockage de ces informations, et à les protéger par le chiffrement (**chapitre 12***).
- Dans la **troisième partie**, **« Machines »**, nous verrons que derrière les informations, il y a toujours des objets matériels : ordinateurs, réseaux, robots, etc. Les premiers ingrédients de ces machines sont des portes booléennes (**chapitre 13**) qui réalisent les fonctions booléennes vues au chapitre 10. Ces portes demandent à être complétées par d'autres circuits, comme les mémoires et les horloges, qui introduisent une dimension temporelle (**chapitre 14**). Nous découvrirons comment fonctionnent les machines que nous utilisons tous les jours (**chapitre 15**). Nous verrons que les réseaux, comme les oignons, s'organisent en couches (**chapitre 16***). Et nous découvrirons enfin les entrailles des robots, que nous apprendrons à commander (**chapitre 17***).
- Dans la **quatrième partie, « Algorithmes »**, nous apprendrons quelques-uns des savoir-faire les plus utiles au XXIe siècle : ajouter des nombres exprimés en base deux (**chapitre 18**), dessiner (**chapitre 19**), retrouver une information par dichotomie (**chapitre 20***), trier des informations (**chapitre 21***) et parcourir un graphe (**chapitre 22***).

Chaque chapitre contient trois types de contenus :

- une partie de **cours** ;
- des sections intitulées **« Savoir-faire »**, qui permettent d'acquérir les capacités essentielles ;
- des **exercices**, avec leur corrigé lorsque nécessaire.

REMARQUE **Chapitres élémentaires et chapitres avancés***

Les chapitres avancés sont notés ici d'un astérisque. Il s'agit des deux ou trois derniers chapitres de chaque partie. Ils sont signalés en début de chapitre.



Exercices difficiles

Les exercices notés d'un cactus sont d'un niveau plus difficile.

Des encadrés « **Aller plus loin** » donnent des ouvertures vers des questions hors-programme. Chaque chapitre se conclut de trois questions de cours sous forme d'encadré intitulé « **Ai-je bien compris ?** ».

Les propositions de projets sont regroupées en fin de manuel.

Parcours possibles

Cet ouvrage peut être parcouru de plusieurs manières. Nous proposons par exemple de commencer par les chapitres élémentaires de la partie **Informations** (7, 8, 9 et 10), de poursuivre par ceux de la partie **Langages** (1, 2 et 3), de continuer par les chapitres avancés de la partie **Informations** (11 et 12), les chapitres élémentaires de la partie **Algorithmes** (18 et 19) et de la partie **Machines** (13, 14 et 15), et enfin de passer aux chapitres avancés de la partie **Machines** (16 et 17), de la partie **Langages** (4, 5 et 6) et de la partie **Algorithmes** (20, 21 et 22).

Il n'est pas nécessaire de lire ces chapitres au même degré de détails. À chaque élève de choisir les thématiques qu'il souhaite approfondir parmi celles proposées, en particulier par le choix de ses projets.

La seule contrainte est d'acquérir assez tôt les bases des langages de programmation, aux chapitres 1, 2 et 3, pour pouvoir écrire soi-même des programmes. Quand on apprend l'informatique, il est en effet important non seulement d'écouter des cours et de lire des livres, mais aussi de mettre en pratique les connaissances que l'on acquiert en écrivant soi-même des programmes, en se trompant et en corrigeant ses erreurs.

Remerciements

Les auteurs tiennent à remercier Ali Assaf, Olivier Billet, Manuel Bricard, Stéphane Bortzmeyer, Alain Busser, David Cachera, Vint Cerf, Julien Cervelle, Sébastion Chapuis, S. Barry Cooper, Ariane Delrocq, Lena Domröse, Raffi Enficiaud, Monique Grandbastien, Guillaume Le Blanc, Fabrice Le Fessant, Philippe Lucaud, Pierre-Étienne Moreau, Claudine Noblet, François Périnet, Gordon Plotkin, François Pottier, David Roche, Laurent Sartre, Dana Scott, Adi Shamir et Joseph Sifakis pour leur aide au cours de la rédaction de ce livre ainsi que l'équipe des éditions Eyrolles, Anne Bougnoux, Laurène Gibaud, Muriel Shan Sei Fan, Gaël Thomas et Camille Vorng pour leur travail éditorial très créatif.

Merci également à Christine Paulin, Raphaël Hertzog, Pierre Néron, Grégoire Péan, Jonathan Protzenko et Dominique Quatravaux pour leurs témoignages vivants.

Première partie

Langages

Dans cette première partie, nous apprenons à écrire des programmes. Pour cela, nous découvrons les ingrédients dont les programmes sont constitués : l'affectation, la séquence et le test (chapitre 1), les boucles (chapitre 2), les types (chapitre 3), les fonctions (chapitre 4*) et les fonctions récursives (chapitre 5*). Pour finir, nous nous penchons sur la notion de langage formel (chapitre 6*).

Dès que l'on commence à maîtriser ces concepts, il devient possible de créer ses propres programmes.

```
int getNombreAleatoire()
{
    return 4;   // Nombre choisi au dé (non truqué)
                // Garanti 100% aléatoire.
}
```

XKCD

Les ingrédients des programmes

1

Un ordinateur peut faire bien des choses, mais il faut d'abord les lui expliquer.

Apprendre la programmation, ce n'est pas seulement savoir écrire un programme, c'est aussi comprendre de quoi il est fait, comment il est fait et ce qu'il fait. Un programme est essentiellement constitué d'expressions et d'instructions.

Nous introduisons dans ce premier chapitre les trois premières instructions fondamentales que sont l'affectation, la séquence et le test. Pour mettre en évidence leur structure, nous indentons les programmes et utilisons les accolades lorsque l'écriture est ambiguë. Nous étudions les instructions en observant les transformations qu'elles opèrent sur l'état de l'exécution du programme, c'est-à-dire sur l'ensemble des boîtes pouvant contenir des valeurs, avec leur nom et leur valeur, le cas échéant.



John Backus (1924-2007) est l'auteur de l'un des premiers langages de programmation : le langage Fortran (1954). Il a par la suite proposé, avec Peter Naur, la *notation de Backus et Naur* qui permet de décrire des grammaires, en particulier celles des langages de programmation (voir le chapitre 6).

Grace Hopper (1906-1992) est, elle aussi, l'auteur d'un des premiers langages de programmation : le langage Cobol (1959). Avant cela, elle a été l'une des premières à programmer le Harvard Mark I de Howard Aiken, l'un des tous premiers calculateurs électroniques.

Un programme est un texte qui décrit un algorithme que l'on souhaite faire exécuter par une machine. Ce texte est écrit dans un langage particulier, appelé *langage de programmation*. Il existe plusieurs milliers de langages de programmation, parmi lesquels Java, C, Python, Caml, Fortran, Cobol, etc. Il n'est cependant pas nécessaire d'apprendre ces langages les uns après les autres, car ils sont tous plus ou moins organisés autour des mêmes notions : déclaration, affectation, séquence, test, boucle, fonction, etc. Ce sont ces notions qu'il importe de comprendre. Dans ce livre, nous utilisons principalement le langage Java. Mais rien de ce que nous disons ici n'est propre à ce langage et les élèves qui en utilisent un autre n'auront aucun mal à transposer.

Nous verrons, au chapitre 3, comment faire pour exécuter un programme sur un ordinateur. Dans ce chapitre et le suivant, nous parlons des programmes de manière théorique, c'est-à-dire sans les exécuter. Bien entendu, les lecteurs sont libres d'anticiper et de lire les premières pages du chapitre 3, s'ils souhaitent exécuter leurs programmes tout de suite.

Un premier programme

Voici un premier petit programme écrit en Java :

```
a = 4;
b = 7;
System.out.println("À vous de jouer");
x = Isn.readInt();
y = Isn.readInt();
if (x == a && y == b) {
  System.out.println("Coulé");}
else {
  if (x == a || y == b) {
    System.out.println("En vue");}
  else {
    System.out.println("À l'eau");}}
```

Quand on exécute ce programme, il affiche `À vous de jouer` puis attend que l'on tape deux nombres au clavier. Si ces deux nombres sont 4 et 7, il affiche `Coulé` ; si le premier est 4 ou le second 7, mais pas les deux, il affiche `En vue`, sinon il affiche `À l'eau`.

Ce programme permet de jouer à la bataille navale, dans une variante simplifiée dans laquelle il n'y a qu'un seul bateau, toujours placé au même endroit et qui n'occupe qu'une seule case de la grille. On considère

qu'un bateau est « en vue » si la case proposée est sur la même ligne ou la même colonne que le bateau.

Exercice 1.1

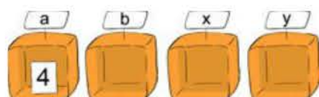
Modifier ce programme afin qu'il affiche `À toi de jouer` et non `À vous de jouer`.

Exercice 1.2

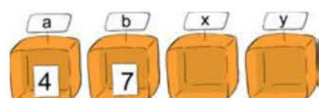
Modifier ce programme afin que le bateau soit sur la case de coordonnées `(6 ; 9)`.

La description du programme

Commençons par observer ce programme pour essayer d'en comprendre la signification. La première ligne contient l'instruction `a = 4;`. Pour comprendre ce qu'il se passe quand on exécute cette instruction, il faut imaginer que la mémoire de l'ordinateur que l'on utilise est composée d'une multitude de petites boîtes. Chacune de ces boîtes porte un nom et peut contenir une valeur. Exécuter l'instruction `a = 4;` a pour effet de mettre la valeur 4 dans la boîte de nom `a`.



Après avoir exécuté cette instruction, on exécute `b = 7;`, ce qui a pour effet de mettre la valeur 7 dans la boîte de nom `b`.



On exécute ensuite l'instruction `System.out.println("À vous de jouer");`, ce qui a pour effet d'afficher à l'écran `À vous de jouer`.

On exécute ensuite l'instruction `x = Isn.readInt();`, ce qui a pour effet d'interrompre le déroulement du programme jusqu'à ce que l'utilisateur tape un nombre au clavier. Ce nombre est alors mis dans la boîte de nom `x`.

DANS D'AUTRES LANGAGES

Texas Instruments et Casio

Ce même algorithme peut s'exprimer dans de nombreux langages. À titre d'exemple, le voici exprimé dans le langage des calculatrices Texas Instruments et Casio. Dans ces deux cas *l'algorithme* utilisé est le même qu'en Java. Les seules différences sont dans la manière d'exprimer cet algorithme : la variable à affecter est située à droite de la flèche pour les calculatrices, l'instruction de test est structurée par des mots-clés supplémentaires, entre autres.

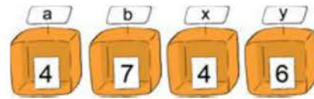
- Texas Instruments

```
PROGRAM:BATAILLE
:4 →A
:7 →B
:Disp "À vous de jouer"
:Input X
:Input Y
:If X = A et Y = B
:Then
:Disp "Coulé"
:Else
:If X = A ou Y = B
:Then
:Disp "En vue"
:Else
:Disp "À l'eau"
:End
:End
```

- Casio

```
======BATAILLE ======
4 →A
7 →B
"À vous de jouer"
? →X
? →Y
If X = A And Y = B
Then "Coulé"
Else
If X = A Or Y = B
Then "En vue"
Else "À l'eau"
IfEnd
IfEnd
```

De même, exécuter l'instruction `y = Isn.readInt();` a pour effet d'interrompre le déroulement du programme jusqu'à ce que l'utilisateur tape un nombre au clavier. Ce nombre est alors mis dans la boîte de nom `y`. À ce point du programme, les boîtes de nom `a`, `b`, `x` et `y` contiennent chacune un nombre. Les nombres 4 et 7 pour les deux premières et les deux nombres entrés au clavier par l'utilisateur pour les deux dernières.



L'exécution du programme doit alors différer selon que les deux nombres donnés par l'utilisateur sont 4 et 7 ou non. Si c'est le cas, on veut afficher `Coulé`, si ce n'est pas le cas on veut faire autre chose. C'est ce que fait l'instruction :

```
if (x == a && y == b) {
  System.out.println("Coulé");}
else {
  if (x == a || y == b) {
    System.out.println("En vue");}
  else {
    System.out.println("À l'eau");}}
```

Exécuter cette instruction a pour effet de calculer la valeur de l'expression booléenne `x == a && y == b`, où le symbole `&&` représente un *et*. Cette valeur est `true` (vrai) quand `x` est égal à `a` et `y` est égal à `b`, ou `false` (faux) quand ce n'est pas le cas. En fonction de la valeur de cette expression, on exécute ou bien l'instruction `System.out.println("Coulé");` ou bien l'instruction :

```
if (x == a || y == b) {
  System.out.println("En vue");}
else {
  System.out.println("À l'eau");}
```

Cette instruction étant de la même forme, son exécution a pour effet de calculer la valeur de l'expression booléenne `x == a || y == b`, où le symbole `||` représente un *ou*, et en fonction de la valeur de cette expression d'exécuter ou bien l'instruction `System.out.println("En vue");` ou bien l'instruction `System.out.println("À l'eau");`.



Exercice 1.3

En C, le même extrait de programme s'écrit ainsi :

```
a = 4;
b = 7;
printf("À vous de jouer\n");
scanf("%d",&x);
scanf("%d",&y);
if (x == a && y == b) {
  printf("Coulé\n");}
else {
  if (x == a || y == b) {
    printf("En vue\n");}
  else {
    printf("À l'eau\n");}}
```

Quelles sont les ressemblances et les différences entre Java et C ?

SAVOIR-FAIRE Modifier un programme existant pour obtenir un résultat différent

L'intérêt de partir d'un programme existant est qu'il n'est pas toujours nécessaire d'en comprendre le fonctionnement en détail pour l'adapter à un nouveau besoin.
Il importe avant tout :

- d'identifier les parties du programme qui doivent être modifiées et celles qui sont à conserver,
- de les modifier en conséquence,
- éventuellement d'adapter les entrées et les sorties au nouveau programme,
- et, comme toujours, de le tester sur des exemples bien choisis.

Exercice 1.4 (avec corrigé)

Le programme suivant permet de calculer le prix toutes taxes comprises d'un article, connaissant son prix hors taxes, dans le cas où le taux de TVA est de 19,6 %.

```
System.out.println("Quel est le prix hors taxes ?");
ht = Isn.readDouble();
ttc = ht + ht * 19.6 / 100.0;
System.out.print("Le prix toutes taxes comprises est ");
System.out.println(ttc);
```

Adapter ce programme pour permettre à l'utilisateur de choisir le taux de TVA.

Même si l'on n'est pas un expert en calcul de pourcentages, on identifie facilement qu'il faut remplacer le `19.6` *de la troisième ligne par un taux quelconque. Pour que ce taux puisse être choisi par l'utilisateur, il doit être stocké dans une nouvelle boîte : appelons-la* `taux`*. Le contenu de cette boîte doit être saisi au*

clavier. Il faut donc prévoir l'entrée correspondante. Voici donc le nouveau programme avec, en gras, les éléments ajoutés ou modifiés :

```
System.out.println("Quel est le prix hors taxes ?");
ht = Isn.readDouble();
System.out.println("Quel est le taux de TVA ?");
taux = Isn.readDouble();
ttc = ht + ht * taux / 100.0;
System.out.print("Le prix toutes taxes comprises est ");
System.out.println(ttc);
```

Exercice 1.5

En général, à la bataille navale, un bateau n'est « en vue » que si la case touchée est immédiatement voisine de celle du bateau. Modifier le premier programme de ce chapitre pour tenir compte de cette règle. On pourra traiter le cas où les cases diagonalement adjacentes au bateau sont « en vue » et le cas où elles ne le sont pas.

Les ingrédients d'un programme

Le programme de bataille navale utilise des instructions de différentes formes :

- des *affectations* de la forme `v = e;` où `v` est une variable et `e` une expression,
- des *instructions d'entrée* de la forme `v = Isn.readInt();` où `v` est une variable,
- des *instructions de sortie* de la forme `System.out.println(e);` où `e` est une expression,
- des *séquences* de la forme `p q` (c'est-à-dire `p` suivi de `q`) où `p` et `q` sont deux instructions,
- des *tests* de la forme `if (e) p else q` où `e` est une expression et `p` et `q` deux instructions.

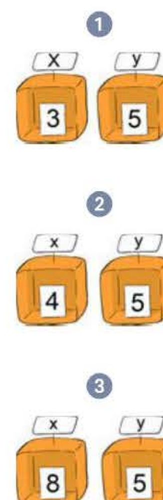
État de l'exécution d'un programme

On appelle *état de l'exécution d'un programme* le triplet formé par le nombre de boîtes utilisées, le nom de chacune d'elles et la valeur qu'elle contient.

La mémoire de l'ordinateur est constituée d'une multitude de petites boîtes, un programme n'utilise en général que quelques-unes de ces boîtes. Chaque boîte utilisée par le programme a un nom et contient une valeur. On appelle *état de l'exécution d'un programme*, le triplet formé par le nombre de boîtes, le nom de chacune d'elles et la valeur qu'elle contient.

Exécuter une instruction a pour effet de transformer cet état.

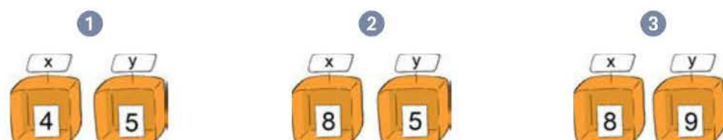
- Exécuter l'affectation `v = e;` a pour effet de calculer la valeur de l'expression `e` dans l'état courant et de modifier cet état en mettant cette valeur dans la boîte de nom `v`. Par exemple, exécuter l'instruction `x = 4;` dans l'état ❶ produit l'état ❷ (voir ci-contre). De même, exécuter l'instruction `x = y + 3;` dans l'état ❶ produit l'état ❸.



- Exécuter l'instruction d'entrée `v = Isn.readInt();` où `v` est une variable a pour effet d'interrompre le déroulement du programme jusqu'à ce que l'utilisateur tape un nombre au clavier. Ce nombre est alors mis dans la boîte de nom `v`. Des instructions similaires, `v = Isn.readDouble();` et `v = Isn.readString();` permettent à l'utilisateur de taper un nombre à virgule ou une chaîne de caractères.
- Exécuter l'instruction de sortie `System.out.print(e);` où `e` est une expression ne modifie pas l'état, mais a pour effet de calculer la valeur de l'expression `e` dans l'état courant et d'afficher cette valeur à l'écran. Exécuter l'instruction de sortie `System.out.println(e);` affiche la valeur de l'expression `e` puis passe à la ligne. Exécuter l'instruction de sortie `System.out.println();` n'affiche rien mais passe à la ligne.
- Exécuter la séquence `p q` où `p` et `q` sont deux instructions a pour effet d'exécuter `p` puis `q` dans l'état obtenu. Par exemple, exécuter l'instruction :

```
x = 8;
y = 9;
```

dans l'état ❶ exécute l'instruction `x = 8;` ce qui produit l'état ❷ puis l'instruction `y = 9;` ce qui produit l'état ❸.



- Exécuter le test `if (e) p else q` où `e` est une expression et `p` et `q` sont deux instructions a pour effet de calculer la valeur de l'expression `e`, puis d'exécuter l'instruction `p` ou l'instruction `q`, selon que la valeur de `e` est `true` (vrai) ou `false` (faux). Par exemple, exécuter l'instruction :

```
if (x < 7) {
  System.out.println("un peu");}
else {
  System.out.println("beaucoup");}
```

COMPRENDRE Une instruction composée mais unique

Attention, l'instruction :

```
x = 8;
y = 9;
```

est une unique instruction, à savoir une séquence de deux instructions plus petites :

```
x = 8;
```

et

```
y = 9;
```



dans l'état 1 affiche `un peu`, car la valeur de l'expression `x < 7` dans cet état est `true`. En revanche, exécuter cette instruction dans l'état 2 affiche `beaucoup`.

Une variante du test est le test *sans* `else` : `if (e) p` où `e` est une expression et `p` est une instruction. Exécuter cette instruction a pour effet de calculer la valeur de l'expression `e`, puis d'exécuter l'instruction `p` si la valeur de `e` est `true`. Par exemple, exécuter l'instruction :

```
if (x < 7) {
  System.out.println("un peu");}
```

dans l'état 1 affiche `un peu`, alors qu'exécuter cette instruction dans l'état 2 n'affiche rien.

SAVOIR-FAIRE Comprendre un programme et expliquer ce qu'il fait

Identifier le rôle de chacune des variables utilisées. Si nécessaire, dérouler à la main une exécution du programme en notant l'état de l'exécution du programme au fur et à mesure.

Exercice 1.6 (avec corrigé)

Que fait ce programme ?

```
a = Isn.readInt();
b = Isn.readInt();
c = Isn.readInt();
d = Isn.readInt();
if (b == 0 || d == 0) {
  System.out.println("Dénominateur nul interdit !");}
else {
  System.out.println(a * c);
  System.out.println(b * d);}
```

Il y a ici quatre entrées `a`, `b`, `c` *et* `d`, *et deux sorties qui correspondent aux produits* `a * c` *et* `b * d`. *Le premier test indique que ni* `b` *ni* `d` *ne doivent être nulles. De tous ces éléments, on déduit que les entrées représentent sans doute les fractions* `a / b` *et* `c / d`, *que le programme calcule le produit de ces deux fractions, lorsqu'elles existent, et donne à nouveau le résultat sous la forme d'une fraction. On notera que ce qui peut rendre ce programme difficile à lire est, entre autres choses, les noms peu parlants choisis pour les variables. On gagnerait ainsi à renommer* `a` *en* `numerateur1`, `b` *en* `denominateur1`, `c` *en* `numerateur2`, *et* `d` *en* `denominateur2`.

Exercice 1.7

Que fait ce programme ? Comment devrait-on renommer ses variables ?

```
a = Isn.readInt();
b = Isn.readInt();
c = Isn.readInt();
```

```
d = Isn.readInt();
if (b == 0 || d == 0) {
  System.out.println("Dénominateur nul interdit !");}
else {
  System.out.println(a * d + c * b);
  System.out.println(b * d);}
```

Exercice 1.8

L'exécution de l'instruction :

```
x = 4;
y = x + 1;
x = 10;
System.out.println(y);
```

produit-elle l'affichage de la valeur 5 ou de la valeur 11 ?

SAVOIR-FAIRE Écrire un programme

Identifier les entrées et les sorties du programme et, dans la mesure du possible, les variables intermédiaires dont on aura besoin. Si le programme doit « prendre une décision », une ou plusieurs instructions de tests sont nécessaires.

Exercice 1.9 (avec corrigé)

Écrire un programme qui, étant donné une équation du second degré, détermine le nombre de ses solutions réelles et leurs valeurs éventuelles.

L'entrée est une équation du second degré `a x² + b x + c = 0`*, fournie sous la forme de ses coefficients* `a`*,* `b` *et* `c`*. La sortie sera l'affichage du nombre de solutions réelles et de leurs valeurs. Le rôle du discriminant* $\Delta = b^2 - 4ac$ *est ici suffisamment important pour mériter une variable intermédiaire* `delta` *qui stocke sa valeur.*

Il faut distinguer trois cas selon le signe du discriminant, ce qui se fait bien entendu à l'aide de tests.

```
a = Isn.readDouble();
b = Isn.readDouble();
c = Isn.readDouble();
delta = b * b - 4 * a * c;
if (delta < 0.0) {
  System.out.println("Pas de solution");}
else {
  if (delta == 0.0) {
    System.out.print("Une solution : ");
    System.out.println(- b / (2 * a));}
  else {
    System.out.print("Deux solutions : ");
    System.out.print((- b - Math.sqrt(delta)) / (2 * a));
    System.out.print(" et ");
    System.out.println((- b + Math.sqrt(delta)) / (2 * a));}}
```

Exercice 1.10

Essayer le programme ci-dessus avec les entrées a = 1.0, b = 0.0, c = 1.0E-10 et a = 1.0, b = 0.0, c = -1.0E-10. Montrer qu'une infime variation sur l'un des coefficients permet de franchir la ligne qui sépare les cas où l'équation a des solutions des cas où elle n'en a pas.

Essayer le programme ci-dessus avec les entrées a = 1.0, b = 6.0, c = 9.0 et a = 0.1, b = 0.6, c = 0.9. Expliquer les résultats.

SAVOIR-FAIRE Mettre un programme au point en le testant

Pour vérifier si un programme ne produit pas d'erreur au cours de son exécution et s'il effectue réellement la tâche que l'on attend de lui, une première méthode consiste à exécuter plusieurs fois ce programme, en lui fournissant des entrées, appelées *tests*, qui permettent de détecter les erreurs éventuelles. Pour qu'elles jouent leur rôle, il faut choisir ces entrées de sorte que :

- on sache quelle doit être la sortie correcte du programme avec chacune de ces entrées,
- chaque cas distinct d'exécution du programme soit parcouru avec au moins un choix d'entrées,
- les cas limites soient essayés : par exemple le nombre 0, la chaîne vide ou à un seul caractère.

Exercice 1.11 (avec corrigé)

Proposer un jeu de tests satisfaisant pour le programme de bataille navale.

Au minimum, il faut vérifier que le bateau est effectivement coulé si l'on donne les bonnes coordonnées, mais non coulé si l'on en donne de mauvaises. Par ailleurs, il faut tester si le programme affiche correctement `En vue`, *et donc tester au moins une case dans la même colonne que le bateau et une case dans la même ligne. Ces deux derniers tests permettront également de vérifier que les instructions conditionnelles du programme sont écrites correctement, et que par exemple il ne suffit pas d'avoir trouvé la bonne ligne pour couler le bateau. On testera donc le programme sur les entrées suivantes, par exemple, avec les résultats attendus :*

- *(4 ; 7) :* `Coulé`,
- *(1 ; 2) :* `À l'eau`,
- *(4 ; 9) :* `En vue` *(même ligne),*
- *(8 ; 7) :* `En vue` *(même colonne).*

On pourrait également tester ce qu'il se passe si l'on entre une coordonnée décimale ou une coordonnée qui dépasse les limites du tableau de jeu.

Exercice 1.12

Proposer un jeu de tests satisfaisant pour le programme de calcul des solutions réelles d'une équation du second degré ci-avant.

Les instructions et les expressions

L'affectation `x = y + 3;` est une instruction. Elle est composée d'une variable `x` et d'une *expression* `y + 3`.

On attribue une *valeur* à chaque expression. Pour les expressions sans variables, comme `(2 + 5) * 3`, dont la valeur est 21, la valeur s'obtient simplement en effectuant les opérations présentes dans l'expression, dans cet exemple une addition et une multiplication. La valeur d'une expression qui contient des variables, par exemple `(2 + x) * 3`, se définit de la même manière, mais dépend de l'état dans lequel on calcule cette valeur.

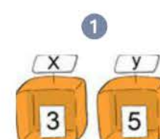
Par exemple, la valeur de l'expression `(2 + x) * 3` dans l'état ❶ est 15, alors que celle de cette même expression dans l'état ❷ est 18.

Une expression

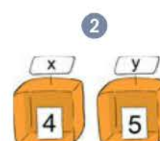
Une *expression* est un élément de programme qui peut être de différentes formes :

- une variable, par exemple `x`, `y` ;
- une constante, par exemple, `3`, `3.14`, `6.02E23`, `true`, `"Coulé"` ;
- une expression de la forme `e + e'`, `e * e'`, `e == e'` formée d'une opération comme `+`, `*`, `==` et d'une ou plusieurs expressions, par exemple `y + 3`.

❶



❷





Exercice 1.13

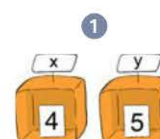
Les suites de symboles suivantes sont-elles des instructions ou des expressions ?

- `x`
- `x = y;`
- `x = y + 3;`
- `x + 3;`
- `System.out.println(x + 3);`
- `x = Isn.readInt();`
- `x == a`
- `x == a && y == b`

Exercice 1.14

Déterminer la valeur des expressions suivantes dans l'état ❶.

❶



- `y + 3`
- `x + 3`
- `x + y`
- `x * x`
- `y == 5`
- `x == 3 && y == 5`



Exercice 1.15

Déterminer dans chacun des cas suivants tous les états tels que :

- `y - 2` vaut 1,
- `x * x` vaut 4,
- `x + y` vaut 1,
- ces trois conditions à la fois.

Les opérations

Les expressions sont formées en utilisant les opérations suivantes :

`+`	Addition entière
`-`	Soustraction entière
`*`	Multiplication entière
`/`	Quotient de la division euclidienne. Attention cette division est inhabituelle pour les nombres négatifs : `-5 / 2` vaut -2 et non -3, et `-5 / -2` vaut 2.
`%`	Reste de la division euclidienne. Attention encore aux nombres négatifs : `-5 % 2` vaut -1 et `-5 % -2` aussi.
`+`	Addition décimale
`-`	Soustraction décimale
`*`	Multiplication décimale
`/`	Division décimale
`Math.pow`	Puissance
`Math.sqrt`	Racine
`Math.PI`	π
`Math.sin`	Sinus
`Math.cos`	Cosinus
`Math.exp`	Exponentielle
`Math.log`	Logarithme népérien
`Math.abs`	Valeur absolue
`Math.min`	Minimum
`Math.max`	Maximum
`Math.floor`	Partie entière
`Math.random`	Nombre aléatoire décimal entre 0 et 1, selon la loi uniforme
`==`	Égal. S'applique aux valeurs numériques et booléennes, mais pas aux chaînes de caractères ni aux tableaux.
`!=`	Différent. S'applique aux valeurs numériques et booléennes, mais pas aux chaînes de caractères ni aux tableaux.

`<=`	Inférieur ou égal.
`<`	Inférieur strictement.
`>=`	Supérieur ou égal.
`>`	Supérieur strictement.
`Isn.stringEqual`	Prend en argument deux chaînes de caractères et renvoie la valeur `true` si ces deux chaînes sont égales et la valeur `false` sinon.
`Isn.stringAlph`	Prend en argument deux chaînes de caractères et renvoie la valeur `true` si la première chaîne est avant la seconde dans l'ordre alphabétique, et la valeur `false` sinon.
`Isn.stringLength`	Prend en argument une chaîne de caractères et renvoie un entier qui est sa longueur.
`Isn.stringNth`	Prend en argument une chaîne de caractères `s` et un entier `n`, compris entre 0 et la longueur de la chaîne moins 1, et renvoie la chaîne de caractères formée d'un seul caractère qui est le `n`-ième de la chaîne `s`.
`Isn.asciiString`	Prend en argument un entier `n` et retourne une chaîne de caractères qui contient un unique caractère dont le code ASCII (voir le chapitre 8) est `n`.
`Isn.stringCode`	Fonction inverse de la précédente, qui prend en argument une chaîne de caractères `s` et retourne le code ASCII du premier caractère de cette chaîne.
`+`	Concaténation. S'applique à deux chaînes de caractères et construit une unique chaîne formée de la première, suivie de la deuxième.
`!`	Non.
`&`	Et (Variante : `&&`.)
`|`	Ou (Variante : `||`.)

ALLER PLUS LOIN Les opérations && et ||

L'opération && est une variante de &, telle que la valeur de l'expression t && u soit false quand la valeur de t est false, même si la valeur de l'expression u n'est pas définie.
De même, l'opération || est une variante de | telle que la valeur de l'expression t || u soit true quand la valeur de t est true, même si la valeur de l'expression u n'est pas définie.

Ainsi l'exécution de l'instruction :

```
System.out.println(x != 0 & 1/x > 2);
```

provoque une erreur, quand x est égal à 0, mais celle de l'instruction :

```
System.out.println(x != 0 && 1/x > 2);
```

affiche false.



Exercice 1.16

Le but de cet exercice est d'écrire un programme qui demande à l'utilisateur une date comprise entre le 1er janvier 1901 et le 31 décembre 2099 et qui indique le nombre de jours écoulés entre le premier janvier 1901 et cette date.

Une bonne approximation de ce nombre est `(a - 1901) * 365 + (m - 1) * 30 + j - 1;`. Mais il faut lui ajouter deux termes correctifs. Le premier est dû au fait que tous les mois ne font pas trente jours. On peut montrer que ce terme correctif vaut `m / 2` quand m est compris entre 1 et 2 et vaut `(m + m / 8) / 2 - 2` quand m est compris entre 3 et 12. Le second est dû aux années bissextiles. On peut montrer que ce terme correctif vaut `(a - 1900)/4 - 1` si a est un multiple de 4 et m est compris entre 1 et 2 et vaut `(a - 1900)/4` sinon.

- Écrire un programme qui demande à l'utilisateur trois nombres qui constituent une date comprise entre le premier janvier 1901 et le 31 décembre 2099, par exemple 20 / 12 / 1996, et qui indique le nombre de jours écoulés entre le premier janvier 1901 et cette date.
- Écrire un programme qui demande à l'utilisateur deux dates et indique le nombre de jours écoulés entre ces deux dates.
- Sachant que le premier janvier 1901 était un mardi, écrire un programme qui demande à l'utilisateur une date et indique le jour de la semaine correspondant à cette date.

Exercice 1.17

En utilisant la fonction Math.random, écrire un programme qui simule la loi uniforme sur l'intervalle [*a* ; *b*], où *a* et *b* sont deux réels donnés.

Exercice 1.18

En utilisant la fonction Math.random, écrire un programme qui affiche aléatoirement pile ou face de façon équiprobable.

ALLER PLUS LOIN Les ordinateurs et le hasard

Parmi les opérations de base, nous avons cité la fonction Math.random, qui renvoie un nombre aléatoire compris entre 0 et 1. Si l'on s'y arrête quelques secondes, l'existence d'une telle fonction est contradictoire avec la notion même d'algorithme : un processus suffisamment bien décrit et détaillé pour être exécuté sans erreur ni initiative de la part d'une machine ne peut pas mener à un résultat imprévisible et différent à chaque exécution. Pourtant l'introduction de hasard dans les programmes est indispensable, par exemple pour créer des situations imprévues dans les logiciels de jeux, mais aussi pour résoudre certains problèmes qui ne peuvent pas être résolus sans une part de hasard, comme on le verra au chapitre 16. La fonction Math.random ne génère pas de nombres réellement aléatoires, mais les résultats obtenus sont suffisamment proches de tirages aléatoires pour la plupart des applications qui utilisent ce genre de nombres. L'exercice 2.8 donne un exemple d'un tel générateur de nombres pseudo-aléatoires.

Les accolades

L'expression `x - y + z` peut être construite ou bien avec le signe + et les deux expressions `x - y` et z, ou alors avec le signe - et les deux expressions x et `y + z`. Comme en mathématiques, on lève cette ambiguïté en utilisant des parenthèses : on écrit la première expression `(x - y) + z` et

la seconde `x - (y + z)`. Et, comme en mathématiques, on décide que, quand on n'utilise pas de parenthèses, l'expression `x - y + z` signifie `(x - y) + z` et non `x - (y + z)`.

Un problème similaire se pose avec les instructions : l'instruction `if (x == 0) y = 1; else y = 2; z = 4;` peut être construite ou bien comme la séquence des instructions `if (x == 0) y = 1; else y = 2;` et `z = 4;` ou alors comme le test formé de l'expression `x == 0` et des instructions `y = 1;` et `y = 2; z = 4;`

On lève cette ambiguïté en utilisant, non des parenthèses, mais des accolades, et on écrit `{if (x == 0) y = 1; else y = 2;} z = 4;` la première instruction et `if (x == 0) y = 1; else {y = 2; z = 4;}` la seconde. Comme dans le cas des expressions, on décide que, quand on n'utilise pas d'accolades, l'instruction `if (x == 0) y = 1; else y = 2; z = 4;` signifie `{if (x == 0) y = 1; else y = 2;} z = 4;` et non `if (x == 0) y = 1; else {y = 2; z = 4;}`. Le test est prioritaire sur la séquence.

Dans un test `if (b) p else q`, la coutume est de mettre toujours les expressions `p` et `q` entre accolades, afin de faciliter la lecture des programmes :

```
if (x == 0) {
  y = 1;}
else {
  y = 2;
  z = 4;}
```

Enfin, on peut aussi mettre des accolades autour de zéro instruction. L'instruction obtenue, `{}`, signifie « ne rien faire ». On verra au chapitre 17 une utilisation de cette instruction.

Exercice 1.19

Quel est le résultat de l'exécution des instructions :

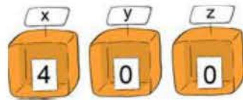
```
if (x == 4) {
  y = 1;}
else {
  y = 2;
  z = 3;}
```

```
{
  if (x == 4) {
    y = 1;}
  else {
    y = 2;}}
z = 3;
```

et

```
if (x == 4) {
  y = 1;}
else {
  y = 2;}
z = 3;
```

dans l'état :





Exercice 1.20

Le but de cet exercice est de montrer que si l'on exécute l'instruction `{p q} r` dans un état `e` on obtient le même résultat que si l'on exécute l'instruction `p {q r}`. Soit e_2 l'état obtenu en exécutant l'instruction `p` dans l'état `e`, e_3 l'état obtenu en exécutant l'instruction `q` dans l'état e_2 et e_4 l'état obtenu en exécutant l'instruction `r` dans l'état e_3.

1. Quel est l'état produit par l'exécution de l'instruction `p q` dans l'état `e` ?
2. Quel est l'état produit par l'exécution de l'instruction `{p q} r` dans l'état `e` ?
3. Quel est l'état produit par l'exécution de l'instruction `q r` dans l'état e_2 ?
4. Quel est l'état produit par l'exécution de l'instruction `p {q r}` dans l'état `e` ?

SAVOIR-FAIRE Indenter un programme

Indenter un programme, c'est-à-dire insérer des espaces blancs en début de ligne, est un moyen de visualiser le niveau d'imbrication auquel une instruction se trouve. On insère un espace blanc par accolade ouverte et non fermée située avant cette instruction dans le programme. Dans certains langages, comme Python, l'indentation joue le rôle de parenthèses, comme les accolades en Java. En Java, en revanche, l'indentation est une aide à la lecture, mais ne change pas la signification des programmes.

Exercice 1.21 (avec corrigé)

Écrire un programme qui affiche le tarif du timbre à poser sur une lettre en fonction de son type et de son poids.

On trouve sur le site web de la Poste le tableau suivant (au 1er octobre 2011) :

Poids jusqu'à	Lettre verte	Lettre prioritaire	Ecopli
20 g	0,57 €	0,60 €	0,55 €
50 g	0,95 €	1,00 €	0,78 €
100 g	1,40 €	1,45 €	1,00 €

On peut par exemple considérer que les différentes gammes de poids sont des sous-cas dans chaque type de lettre. Le programme est donc constitué de trois tests correspondant aux différents types de lettres, l'instruction exécutée dans chacun des cas étant elle-même constituée de trois tests correspondant aux différents poids. On décide de ne rien afficher quand les entrées `type` *et* `poids` *ne correspondent pas à une catégorie du tableau.*

```
if (Isn.stringEqual(type,"verte")) {
  if (poids <= 20) {
    System.out.println(0.57);}
  else {
    if (poids <= 50) {
      System.out.println(0.95);}
    else {
      if (poids <= 100) {
        System.out.println(1.40);}}}}
else {
  if (Isn.stringEqual(type,"prioritaire")) {
    if (poids <= 20) {
      System.out.println(0.60);}
    else {
      if (poids <= 50) {
        System.out.println(1.00);}
      else {
        if (poids <= 100) {
          System.out.println(1.45);}}}}
  else {
    if (Isn.stringEqual(type,"ecopli"))
      if (poids <= 20) {
        System.out.println(0.55);}
      else {
        if (poids <= 50) {
          System.out.println(0.78);}
        else {
          if (poids <= 100) {
            System.out.println(1.00);}}}}}
```

Exercice 1.22

Indenter les programmes les suivants :

- `if (x == 4) {y = 1;} else {y = 2; z = 3;}`
- `if (x == 4) {y = 1;} else {y = 2;} z = 3;`.

Ai-je bien compris ?

- Quelles sont les trois instructions présentées dans ce chapitre et qui permettent de construire un programme élémentaire ?
- Quelle est la différence entre une instruction et une expression ?
- Comment l'exécution d'une instruction transforme-t-elle l'état de l'exécution du programme ?

Les boucles 2

Un ordinateur est fait pour effectuer des calculs longs et répétitifs.

Dans ce chapitre, nous introduisons une nouvelle instruction, la boucle, qui permet d'exécuter une instruction plusieurs fois. Nous en présentons deux variantes, la boucle `for` et la boucle `while`. Nous expliquons comment manipuler le compteur d'une boucle `for`, dans quels cas plutôt utiliser une boucle `while`, et pourquoi il peut arriver que l'exécution d'une boucle ne s'arrête jamais.



Gilles Kahn (1946-2006) et **Gordon Plotkin** (1946-) ont proposé des outils pour décrire la *sémantique* des langages de programmation, c'est-à-dire ce qu'il se passe quand on exécute un programme. Gilles Kahn est aussi l'auteur, avec Gérard Huet, du système *Mentor*, l'un des premiers systèmes qui permet de définir de manière complètement formelle des langages de programmation. On doit à Gordon Plotkin des contributions à de nombreux domaines de l'informatique, en particulier la démonstration automatique et la théorie des systèmes concurrents.

Exemple Certaines instructions sont exécutées plusieurs fois

Pour écrire un programme qui affiche le calendrier :

1 janvier
2 janvier
3 janvier
4 janvier
5 janvier
6 janvier
7 janvier
8 janvier
9 janvier
10 janvier
11 janvier
12 janvier
13 janvier
14 janvier
15 janvier
16 janvier
17 janvier
18 janvier
19 janvier
20 janvier
21 janvier
22 janvier
23 janvier
24 janvier
25 janvier
26 janvier
27 janvier
28 janvier
29 janvier
30 janvier
31 janvier

nous ne voulons pas écrire, dans le programme, l'instruction `System.out.println` trente et une fois, mais *écrire* cette instruction une seule fois et faire en sorte qu'elle soit *exécutée* trente et une fois au cours de l'exécution du programme.

Au cours de l'exécution d'un programme construit avec les instructions présentées au chapitre 1, chaque instruction du programme est exécutée au plus une fois. Par exemple, au cours de l'exécution de l'instruction :

```
x = 1;
if (x == 2) {
  y = 3;}
else {
  y = 7;}
```

l'instruction `y = 7;` est exécutée une fois et l'instruction `y = 3;` n'est pas exécutée, mais aucune instruction n'est exécutée plusieurs fois. Or, bien souvent, nous voulons effectuer des calculs dans lesquels certaines instructions sont exécutées plusieurs fois (voir l'exemple ci-contre).

Permettre à une instruction d'être exécutée plusieurs fois au cours de l'exécution d'un programme est le but d'une nouvelle instruction : la *boucle*.

La boucle for

La première forme de boucle, présentée dans ce chapitre, est la boucle `for`. C'est une instruction de la forme `for (i = e; i <= e'; i = i + 1) p` où `i` est une variable, `e` et `e'` sont des expressions et `p` est une instruction, appelée *corps* de cette boucle. Comme dans le cas des tests, la coutume est de toujours mettre le corps d'une boucle entre accolades. Exécuter la boucle `for (i = e; i <= e'; i = i + 1) p` a pour effet d'exécuter l'instruction `p` $(n - m + 1)$ fois, où m est la valeur de l'expression `e` et n celle de l'expression `e'`, dans des états dans lesquels la valeur de la variable `i` est successivement m, $m + 1$, …, n.

Par exemple, exécuter la boucle :

```
for (i = 1; i <= 10; i = i + 1) {
  System.out.print("allô ");}
System.out.println("t'es où ?");
```

a pour effet d'afficher :

```
allô allô allô allô allô allô allô allô allô allô t'es où ?
```

Exécuter la boucle :

```
for (i = 1; i <= 10; i = i + 1) {
  System.out.println(i);}
```

a pour effet d'afficher ①.

Cette description de ce qu'il se passe quand on exécute une boucle n'est exacte que quand deux conditions sont vérifiées : d'une part, la variable `i` ne doit pas être affectée au cours de l'exécution du corps `p` de la boucle, d'autre part, la valeur de l'expression `e'` ne doit pas changer au cours de cette exécution. Par exemple, exécuter l'instruction :

```
for (i = 1; i <= 10; i = i + 1) {
  System.out.println(i);}
```

affiche ❶, ce qui montre que le corps de la boucle est bien exécuté dix fois.

Mais exécuter l'instruction :

```
for (i = 1; i <= 10; i = i + 1) {
  System.out.println(i);
  i = i + 1;}
```

où la valeur de la variable `i` est augmentée de 1 à chaque fois que l'instruction `p` est exécutée affiche ❷, ce qui montre que le corps de la boucle n'est exécuté que cinq fois.

Et, de même, exécuter l'instruction :

```
x = 10;
for (i = 1; i <= x; i = i + 1) {
  System.out.print(i);
  System.out.print(" ");
  System.out.println(x);
  x = x - 1;}
```

où la valeur de la variable `x`, qui apparaît dans l'expression `e'`, est diminuée de 1 à chaque fois que l'instruction `p` est exécutée affiche ❸, ce qui montre que le corps de la boucle n'est exécuté que cinq fois également.

On retiendra de ces deux exemples, qu'il ne faut pas, dans le corps d'une boucle, affecter la variable `i` ni aucune des variables qui apparaissent dans l'expression `e'`. Quand on est tenté de le faire, cela est souvent le signe qu'il aurait mieux valu utiliser une boucle `while` présentée ci-après.

En utilisant une boucle `for`, on peut maintenant écrire une instruction qui affiche le calendrier ci-avant :

```
for (jour = 1; jour <= 31; jour = jour + 1) {
  System.out.print(jour);
  System.out.println(" janvier");}
```

❶

```
1
2
3
4
5
6
7
8
9
10
```

❷

```
1
3
5
7
9
```

❸

```
1 10
2 9
3 8
4 7
5 6
```

DANS D'AUTRES LANGAGES

Texas Instruments et Casio

Dans le langage des calculatrices Texas Instruments et Casio les boucles s'écrivent ainsi :

- Texas Instruments

```
PROGRAM:COMPTE
:For (I,1,10)
:Disp I
:End
```

- Casio

```
======COMPTE ======
For 1 →I to 10
I ◢
Next
```

SAVOIR-FAIRE Écrire un programme utilisant une boucle for

Identifier si le compteur doit jouer un rôle dans le corps de la boucle. Écrire le corps de la boucle. Prévoir une initialisation des variables en amont de la boucle et un post-traitement en aval.

Exercice 2.1 (avec corrigé)

Écrire un programme qui recueille au clavier les températures de 7 jours successifs et calcule la température moyenne de la semaine.

Ici, le compteur de boucle `jour` *ne représente que le numéro du jour dans la semaine et n'intervient pas dans les calculs. Dans le corps de la boucle, on se contente donc de lire les températures au clavier dans une variable* `temperature` *et de faire la somme des nombres entrés au fur et à mesure dans une variable* `somme`*. L'instruction correspondante sera donc :*

```
temperature = Isn.readDouble();
somme = somme + temperature;
```

Pour que la somme calculée soit correcte, il faut penser à initialiser la variable `somme` *à zéro avant la boucle. Enfin, une fois les 7 températures entrées, il faut convertir cette somme en moyenne et l'afficher :*

```
somme = 0;
for (jour = 1; jour <= 7; jour = jour + 1) {
  temperature = Isn.readDouble();
  somme = somme + temperature;}
System.out.println(somme / 7.0);
```

Exercice 2.2

Modifier le programme précédent pour que l'utilisateur puisse préciser le nombre de jours avant de donner les températures.

Exercice 2.3

Écrire un programme qui calcule et affiche la liste des diviseurs d'un nombre entier naturel entré au clavier.

SAVOIR-FAIRE Imbriquer deux boucles

Quand l'instruction à exécuter à l'intérieur d'une boucle est elle aussi répétitive, le corps de cette boucle contient une seconde boucle et on dit que ces deux boucles sont imbriquées. Les bornes de la boucle interne dépendent souvent du compteur de la boucle externe.

Exercice 2.4 (avec corrigé)

Écrire un programme qui affiche un calendrier pour une année entière.

Ce programme doit avoir une structure en boucles imbriquées : une année est constituée de douze mois et chaque mois est à son tour constitué de plusieurs

jours. Si tous les mois de l'année avaient trente jours, il suffirait d'écrire le programme suivant :

```
for (mois = 1; mois <= 12; mois = mois + 1) {
  for (jour = 1; jour <= 30; jour = jour + 1) {
    System.out.print(jour);
    System.out.print(" / ");
    System.out.println(mois);}}
```

Mais comme les mois ont un nombre de jours variable, il faut, pour chaque mois, d'abord calculer le nombre de jours `nbj` *du mois en fonction de* `mois`*, puis utiliser une boucle dont l'indice* `jour` *varie de 1 à* `nbj` *:*

```
for (mois = 1; mois <= 12; mois = mois + 1) {
  if (mois == 2) {
    nbj = 28;}
  else {
    nbj = 30 + (mois + mois / 8) % 2;}
  for (jour = 1; jour <= nbj; jour = jour + 1) {
    System.out.print(jour);
    System.out.print(" / ");
    System.out.println(mois);}}
```

Exercice 2.5

Combien de points affiche le programme suivant ?

```
for (i = 1; i <= 100; i = i + 1) {
  System.out.print(".");}
for (j = 1; j <= 100; j = j + 1) {
  System.out.print(".");}
System.out.println();
```

Et celui-ci ?

```
for (i = 1; i <= 100; i = i + 1) {
  for (j = 1; j <= 100; j = j + 1) {
    System.out.print(".");}}
System.out.println();
```

Exercice 2.6

Écrire un programme qui affiche un calendrier pour une année mais en écrivant les mois « janvier », « février », etc. et non 1, 2, etc.



Exercice 2.7

Écrire un programme qui affiche un calendrier qui va du 1er janvier 2001 au 31 décembre 3000. Attention les années multiples de quatre sont bissextiles, sauf les années multiples de cent qui ne le sont pas, sauf les années multiples de quatre cents qui le sont. Ainsi, 2100, 2200, 2300, 2500, 2600, 2700, 2900 et 3000 ne sont pas bissextiles mais 2400 et 2800 le sont.

Ajouter à ce calendrier le nombre de jours écoulés depuis le début du calendrier.

Ajouter à ce calendrier le jour de la semaine.

```
1 lundi 1 janvier 2001
2 mardi 2 janvier 2001
…
365241 mardi 30 décembre 3000
365242 mercredi 31 décembre 3000
```



Exercice 2.8 Fabriquer des nombres pseudo-aléatoires

La suite de nombres définie par récurrence de la manière suivante :

- $u_0 = 13$
- $u_{n+1} = (16805\ u_n + 1)\ \%\ 32768$

semble aléatoire.

1. Écrire un programme qui affiche les 10 000 premiers termes de cette suite.
2. Pour simuler une suite de tirages à pile ou face, on observe le neuvième bit (voir le chapitre 7) de chaque élément de cette suite et on décrète, lors du *i*-ème tirage, que la pièce est tombée du côté pile si le neuvième bit du nombre u_i est un 0, et qu'elle est tombée du côté face si c'est un 1. Écrire un programme qui affiche les 10 000 premiers tirages.
3. On teste la qualité de ce générateur d'aléa en comptant le nombre de fois que la pièce tombe d'un côté et de l'autre. Écrire un programme qui simule 10 000 tirages et compte le nombre de fois que la pièce tombe du côté pile.
4. Qu'obtient-on si on observe le bit des unités au lieu d'observer le neuvième bit ? Expliquer pourquoi : montrer que si u_n est pair alors u_{n+1} est impair et que si u_n est impair alors u_{n+1} est pair.
5. Montrer que, loin d'être réellement aléatoire, la suite *u* est en fait périodique à partir d'un certain rang.

La boucle while

On veut écrire un programme qui prend en argument un nombre à virgule `x` supérieur ou égal à 1 et qui calcule son logarithme entier. Ce programme est formé d'une boucle qui divise `x` par 2 plusieurs fois, jusqu'à obtenir un nombre inférieur ou égal à 1, tout en ajoutant 1 à un nombre `n` à chaque division, pour les compter. Quand la boucle est terminée, la variable `n` contient le nombre recherché. La nouveauté, avec cette boucle, est que, tant que l'exécution du calcul n'est pas achevée, il n'y a aucun moyen de savoir combien de fois le corps de cette boucle sera répété, puisque ce nombre est précisément le nombre que l'on cherche à calculer : le logarithme entier de `x`. Il n'est donc pas possible d'écrire ce programme avec une boucle `for`.

Logarithme entier

On appelle *logarithme entier* elog(*x*) d'un nombre réel *x* supérieur ou égal à 1, le nombre de fois qu'il faut le diviser par deux pour obtenir un nombre inférieur ou égal à 1. Par exemple, le logarithme entier du nombre 60 000 est 16 car 60 000 / 2^{16} = 0.915…, c'est-à-dire 60 000 divisé par 2 seize fois.

C'est pour cela que l'on a introduit, dans les langages de programmation, une autre forme de boucle : la boucle `while`. Dans une telle boucle, le choix

de continuer ou non à répéter le corps de la boucle n'est pas conditionné, comme dans le cas d'une boucle `for`, par un nombre d'itérations fixé avant le début de l'exécution de la boucle, mais il est fait dynamiquement : avant chaque exécution du corps de la boucle, on teste une condition, si cette condition est vérifiée, on exécute le corps de la boucle et on recommence, si elle ne l'est pas, l'exécution de la boucle est achevée.

Une boucle `while` est une instruction de la forme `while (e) p` où `e` est une expression et `p` est une instruction, appelée *corps* de cette boucle. Exécuter la boucle `while (e) p` a pour effet d'exécuter l'instruction `p` plusieurs fois tant que la valeur de l'expression `e` est égale à `true`.

Ainsi, on peut programmer le calcul du logarithme entier d'un nombre `x` de la manière suivante :

```
n = 0;
while (x > 1.0) {
  x = x / 2.0;
  n = n + 1;}
```

DANS D'AUTRES LANGAGES

Texas Instruments et Casio

Dans le langage des calculatrices la boucle `while` s'écrit de la manière suivante :

- Texas Instruments

```
PROGRAM:ELOG
:0 →N
:While X > 1
:X/2 →X
:N+1 →N
:End
```

- Casio

```
======ELOG ======
0 →N
While X > 1
X/2 →X
N+1 →N
WhileEnd
```



Exercice 2.9

Montrer que si $2^{n-1} < x \leq 2^n$, alors $\text{elog}(x) = n$. Montrer que le logarithme entier d'un nombre est son logarithme binaire, défini par $\log_2(x) = \ln(x) / \ln(2)$, arrondi par excès.

SAVOIR-FAIRE Écrire un programme utilisant une boucle while

Identifier la condition. Écrire le corps de la boucle. Prévoir une initialisation des variables en amont de la boucle et un post-traitement en aval.

Exercice 2.10 (avec corrigé)

Rechercher une sous-chaîne dans une chaîne de caractères.

Comme la fonction Rechercher *d'un logiciel de traitement de texte, on cherche si une chaîne de caractères contient, par exemple, la sous-chaîne* `"oui"`*. Pour ce faire on doit tester si les trois caractères aux positions* `n`*,* `n + 1` *et* `n + 2` *de la chaîne* `s` *sont* `"o"`*,* `"u"` *et* `"i"` *jusqu'à ce qu'on trouve ces trois caractères, ou que l'on atteigne la fin de la chaîne.*

```
longueur = Isn.stringLength(s);
n = 0;
while (n <= longueur - 3
       && !(Isn.stringEqual(Isn.stringNth(s,n),"o")
            && Isn.stringEqual(Isn.stringNth(s,n+1),"u")
            && Isn.stringEqual(Isn.stringNth(s,n+2),"i"))) {
          n = n + 1;}
if (n > longueur - 3) {
  System.out.println("pas de oui");}
else {
  System.out.println(n);}
```

Exercice 2.11

On définit une suite u de la manière suivante :

- $u_0 = 1000$
- si $u_n = 1$, alors la suite est finie et u_n est son dernier élément
- si u_n est pair, alors $u_{n+1} = u_n / 2$
- si u_n est impair et distinct de 1, alors $u_{n+1} = 3\ u_n + 1$

Écrire un programme qui affiche les termes de la suite u.

Cette suite est-elle finie ?

Exercice 2.12

Écrire un programme qui détermine le plus petit multiple commun à deux nombres entiers entrés au clavier.

SAVOIR-FAIRE Commenter un programme

Dès que l'on écrit un programme de plus d'une dizaine de lignes, il est indispensable d'ajouter des *commentaires* dans ce programme, autrement dit des lignes écrites en langue naturelle que la machine ne cherche pas à interpréter comme des instructions et qui expliquent le rôle des différentes parties du programme aux codéveloppeurs.

Ces commentaires permettent à un programmeur de comprendre un programme écrit par un autre programmeur ou par lui-même longtemps auparavant.

Pour préciser qu'une ligne est un commentaire, on la fait précéder d'un symbole particulier. En Java, il s'agit de deux barres obliques `//`. Si l'on veut écrire un commentaire sur plusieurs lignes, il faut faire précéder chacune d'entre elles de ce symbole.

Ces commentaires doivent donner des informations supplémentaires sur le sens du programme. Inutile de commenter en disant, par exemple « ceci est une boucle », mais expliquer le rôle de cette boucle.

Exercice 2.13 (avec corrigé)

Reprendre le programme de l'exercice 1.9 qui calcule les solutions d'une équation du second degré et le compléter pour qu'il vérifie que les coefficients entrés au clavier déterminent bien une équation du second degré, autrement dit que `a` est non nul. Commenter le programme ainsi obtenu.

```
// Voici un programme qui résout l'équation du second degré
// a x^2 + b x + c = 0
a = Isn.readDouble();
b = Isn.readDouble();
c = Isn.readDouble();
// Test du coefficient dominant
if (a == 0.0) {
  System.out.println("Pas une équation du second degré");}
else {
  // Calcul du discriminant
  delta = b * b - 4 * a * c;
```

```
  // Affichage des solutions
  if (delta < 0.0){
    System.out.println("Pas de solution");}
  else {
    if (delta == 0.0) {
      System.out.print("Une solution : ");
      System.out.println(- b / (2 * a));}
    else {
      System.out.print("Deux solutions : ");
      System.out.print((- b - Math.sqrt(delta)) / (2 * a));
      System.out.print(" et ");
     System.out.println((- b + Math.sqrt(delta)) / (2 * a));}}}
```

Exercice 2.14

Commenter le programme de la bataille navale.

La non-terminaison

Avec la boucle `while` apparaît un nouveau comportement possible pour les programmes : la *non-terminaison*. Il est possible d'écrire une instruction `while (e) p` telle que la valeur de l'expression `e` soit toujours égale à `true`, si bien que l'exécution de l'instruction `p` se répète et se répète, sans que jamais l'exécution de la boucle ne se termine. Un exemple simple est le suivant :

```
while (true) {
  System.out.print("allô ");}
```

qui affiche `allô allô allô...` sans jamais s'arrêter.

La boucle for, cas particulier de la boucle while

La boucle `while`, qui permet de choisir dynamiquement si l'on continue à répéter le corps de la boucle ou si l'on s'arrête est un outil plus puissant que la boucle `for`. En fait, la boucle `for` est un cas particulier de la boucle `while`.

L'instruction `for (i = e; i <= e'; i = i + 1) p` peut être vue comme une manière plus simple d'écrire l'instruction :

```
i = e;
while (i <= e') {p i = i + 1;}
```

Par exemple, l'instruction :

```
for (i = 1; i <= 10; i = i + 1) {
  System.out.print("allô ");}
```

peut être vue comme une manière plus simple d'écrire l'instruction :

```
i = 1;
while (i <= 10) {
  System.out.print("allô ");
  i = i + 1;}
```

et l'instruction :

```
for (i = 1; i <= 10; i = i + 1) {
  System.out.println(i);}
```

❶

1
2
3
4
5
6
7
8
9
10

peut être vue comme une manière plus simple d'écrire l'instruction :

```
i = 1;
while (i <= 10) {
  System.out.println(i);
  i = i + 1;}
```

qui affiche ❶ également.

Cette explication de ce qu'il se passe quand on exécute une boucle `for` est aussi plus précise que celle que nous avons donnée avant d'introduire la boucle `while`, car elle permet aussi d'expliquer l'exécution d'une boucle dans le corps de laquelle on affecte le compteur `i` ou une variable utilisée dans l'expression `e'`.

Par exemple, l'instruction :

```
for (i = 1; i <= 10; i = i + 1) {
  System.out.println(i);
  i = i + 1;}
```

peut être vue comme une manière plus simple d'écrire l'instruction :

```
i = 1;
while (i <= 10) {
  System.out.println(i);
  i = i + 1;
  i = i + 1;}
```

❷

1
3
5
7
9

Et on comprend donc pourquoi cette instruction affiche ❷.

ALLER PLUS LOIN Savoir si un programme se termine ou non

Savoir si un programme se termine ou non n'est pas une chose facile. Par exemple, quand on exécute l'instruction :

```
s = 4;
p = false;
for (i = 1; i <= s - 1; i = i + 1) {
  j = s - i;
  if (i * i == 25 * j * j) {
    p = true;}}
```

on énumère tous les couples d'entiers strictement positifs `(i ; j)` dont la somme vaut 4, c'est-à-dire les couples (1 ; 3), (2 ; 2), (3 ; 1), et on teste si l'un de ces couples est une solution de l'équation : `i * i == 25 * j * j`.

Comme ce n'est le cas d'aucun de ces trois couples, l'instruction `p = true;` n'est jamais exécutée et la valeur de la variable `p` reste à `false`.

De même, quand on exécute l'instruction :

```
s = 2;
p = false;
while (!p) {
  for (i = 1; i <= s - 1; i = i + 1) {
    j = s - i;
    if (i * i == 25 * j * j) {
      p = true;}}
  s = s + 1;}
```

on énumère tous les couples dont la somme des éléments vaut 2, puis tous les couples dont la somme des éléments vaut 3, et ainsi de suite, puis on teste si l'un de ces couples est une solution de l'équation : `i * i == 25 * j * j`.

Quand on essaiera les couples dont la somme des éléments vaut 6, on essaiera le couple (5 ; 1), qui est une solution de l'équation, on exécutera l'instruction `p = true;` et l'exécution de la boucle `while` se terminera.

Autrement dit, ce programme énumère tous les couples d'entiers strictement positifs `(i ; j)` et se termine quand il trouve une solution de l'équation `i * i == 25 * j * j`.

Comme cette équation a une solution, le programme se termine.

En revanche, si on remplace le nombre `25` par le nombre `2`, le programme énumère tous les couples d'entiers strictement positifs `(i ; j)` et se termine quand il trouve une solution de l'équation `i * i == 2 * j * j`.

Comme cette équation n'a pas de solution, le programme ne se termine pas.

Ces deux programmes sont donc très similaires, puisque l'un cherche les solutions entières de l'équation `i * i == 25 * j * j` et l'autre celles de l'équation `i * i == 2 * j * j` mais l'un se termine et l'autre non.

SAVOIR-FAIRE Choisir entre une boucle for et la boucle while pour écrire un programme

Si on connaît à l'avance le nombre de répétitions à effectuer, la boucle `for` est toute indiquée. À l'inverse, si la décision d'arrêter la boucle ne peut s'exprimer que par un test, c'est la boucle `while` qu'il faut choisir.

Exercice 2.15 (avec corrigé)

Quelle boucle est adaptée à l'écriture de programmes traitant les problèmes suivants :

1. le calcul du total à payer à une caisse enregistreuse,
2. la recherche du jour le plus pluvieux d'une année,
3. le calcul du périmètre d'un polygone,
4. le calcul de la durée d'une émission de radio, connaissant ses horaires de début et de fin ?

1. *Une boucle `while` : on ne sait pas combien il y aura d'articles, on ne s'arrête que lorsque le tapis est vide.*
2. *Une boucle `for` : le corps de la boucle doit être répété 365 fois exactement.*

3. *Cela dépend : si le nombre de côtés est connu, une boucle* `for`*, sinon, une boucle* `while` *qui s'arrête lorsqu'on est revenu au sommet de départ.*
4. *Il n'y a pas besoin de boucle.*

Exercice 2.16

Écrire les programmes proposés dans l'exercice précédent.

Exercice 2.17

Écrire un programme qui affiche un tableau de valeurs pour la fonction `f : x ↦ x² - 2 x - 2`. L'utilisateur choisit les bornes de l'intervalle sur lequel on calcule ces valeurs, ainsi que le pas entre deux valeurs.



Exercice 2.18

Dans cet exercice on écrit plusieurs versions d'un programme qui joue à la bataille navale, autrement dit qui cherche à couler un bateau. Comme dans le premier exemple, on suppose qu'il n'y a qu'un seul bateau d'une seule case, dont la position est connue par l'utilisateur.

1. Programmer l'algorithme naïf qui consiste à essayer toutes les cases systématiquement.
2. Améliorer cet algorithme pour qu'il s'arrête quand il a coulé le bateau.
3. Améliorer cet algorithme pour qu'il cherche le bateau intelligemment si celui-ci est en vue, c'est-à-dire dans une case adjacente au dernier essai effectué.
4. Peut-on encore améliorer cet algorithme pour minimiser le nombre d'essais nécessaires ?

Ai-je bien compris ?

- À quoi sert une boucle ?
- Quelle est la différence entre une boucle `for` et une boucle `while` ?
- Que signifie qu'un programme se termine ?

Les types

3



Robin Milner (1934-2010) est l'auteur de l'un des premiers langages de programmation avec des types polymorphes et implicites : le langage ML. Ce langage est l'ancêtre de nombreux langages contemporains en particulier des langages Caml et Haskell. Par la suite il a développé l'un des premiers langages permettant de décrire des systèmes concurrents, c'est-à-dire formés de plusieurs processus qui s'exécutent en parallèle. Dans son discours de réception du prix Turing, il a insisté sur l'autonomie de l'informatique, qui n'est une partie d'aucune autre science.

Une boîte ne sait pas comment elle s'appelle.
C'est à nous de lui donner un nom et une forme.

Dans ce chapitre, nous introduisons une nouvelle instruction, la déclaration, qui permet d'ajouter une boîte à un état, et nous voyons qu'il y a différents types de boîtes.

Chaque boîte, en fonction de son type, peut contenir un nombre entier, un nombre à virgule, un booléen, une chaîne de caractères… ou plusieurs, dans le cas d'un tableau. Une boîte associée à une variable peut n'exister que pendant l'exécution de certaines parties du programme – on parle de *portée* de la variable. Une instruction qui manipule une variable doit donc se trouver dans la portée de celle-ci, et il est important de comprendre cette notion pour éviter les erreurs. Munis de tous ces ingrédients, nous pouvons enfin exécuter nos programmes.

Pour exécuter le programme donné en début de chapitre 1, il faut d'abord le taper dans un fichier, appelé par exemple `BatailleNavale.java`.

Certains programmes présentés dans ce livre utilisent une extension de Java appelée `Isn`. Il est donc également nécessaire de récupérer le fichier `Isn.java` sur le site de l'éditeur de ce livre et de le mettre dans le même répertoire que ses programmes.

Compléments

- http://www.editions-eyrolles.com/Livre/9782212135435/informatique-et-sciences-du-numerique

Il faut ensuite compiler ce programme, c'est-à-dire le traduire dans le langage propre de l'ordinateur (voir le chapitre 15), avec la commande `javac BatailleNavale.java`. Le programme traduit, qui se trouve alors dans le fichier `BatailleNavale.class`, peut être exécuté par la commande `java BatailleNavale`.

Toutefois, pour que ce programme produise le résultat attendu, il faut l'accompagner de quelques instructions supplémentaires :

```
class BatailleNavale {
  public static void main (String [] args) {
    int a;
    int b;
    int x;
    int y;
    a = 4;
    b = 7;
    System.out.println("À vous de jouer");
    x = Isn.readInt();
    y = Isn.readInt();
    if (x == a && y == b) {
      System.out.println("Coulé");}
    else {
      if (x == a || y == b) {
        System.out.println("En vue");}
      else {
        System.out.println("À l'eau");}}}}
```

On a ajouté six lignes en début de programme et fermé à la fin du programme les accolades ouvertes plus haut.

Déclaration

La *déclaration* `T v; p`, où `T` est un type, `v` une variable et `p` une instruction a pour effet d'ajouter une boîte de nom `v` et de type `T` à l'état, exécuter `p` dans l'état obtenu et supprimer la boîte de nom `v`.

On a vu que ce programme utilise quatre boîtes de noms `a`, `b`, `x` et `y`. Ces boîtes ne sont pas associées de manière permanente à ces noms, mais cette association est créée par les instructions `int a; int b; int x; int y;`, qui s'appellent des *déclarations*. Pour abréger le programme et le rendre plus lisible, on peut remplacer les quatre déclarations par une seule `int a,b,x,y;`.

Les deux premières lignes, quant à elles, constituent un en-tête qui indique que l'on va écrire un programme.

```
class BatailleNavale {
  public static void main (String [] args) {
```

Exercice 3.1

Écrire le programme ci-dessus dans un éditeur. Compiler et exécuter ce programme.

Exercice 3.2

Ouvrir le fichier `BatailleNavale.class` dans un éditeur.

Type d'une variable

Le *type* d'une variable indique la nature des données que cette variable contient : nombre entier, nombre à virgule, chaîne de caractères, etc.

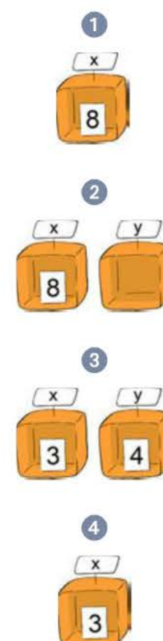
Exécuter l'instruction :

```
int y;
x = 3;
y = 4;
System.out.println(x + y);
```

dans l'état ❶ ajoute à l'état une boîte de nom `y`, ce qui produit l'état ❷ puis exécute l'instruction :

```
x = 3;
y = 4;
System.out.println(x + y);
```

ce qui produit l'état ❸ et affiche `7`, et enfin supprime la boîte de nom `y`, ce qui produit l'état ❹.



Les types de base

En Java, comme dans la plupart des langages de programmation, les expressions sont classées en fonction de leur type :

- les expressions, comme `1 + 2`, dont la valeur est un nombre entier, comme 3, sont de type `int` (*integer* : nombre entier),
- celles comme `1.5 + 1.64`, dont la valeur est un nombre à virgule, comme 3,14, sont de type `double`, ce nom fait référence au fait que les nombres à virgules exprimés sur 32 bits sont appelés nombres à virgule *en simple précision* et ceux exprimés sur 64 bit sont appelés nombres à virgule *en double précision* (voir le chapitre 7),
- celles comme `0 < 2`, dont la valeur est un booléen, `false` (faux) ou `true` (vrai), sont de type `boolean`, dans les langages de programmation, on écrit en général les booléens `false` et `true` et non `0` et `1` pour éviter les confusions avec les nombres,
- celles, comme `"Cou" + "lé"`, dont la valeur est une chaîne de caractères, comme `"Coulé"`, sont de type `String`.

Une valeur de type `int` est exprimée sur 32 bits, selon la méthode vue au chapitre 7, c'est donc un nombre entier relatif compris entre - 2147483648 et 2147483647. Les valeurs de type `double` sont exprimées sur 64 bits selon la méthode vue au chapitre 7 : 1 bit de signe, 11 bits d'exposant, 52 bits de mantisse. Un booléen est simplement un bit : `true` est une autre notation pour le bit 1, et `false` une autre notation pour le bit 0. Dans une valeur de type `String`, chaque caractère est exprimé en Unicode, selon la méthode vue au chapitre 8.

Donner un type à chaque expression a plusieurs avantages : d'une part, cela permet de préciser la manière dont les données doivent être exprimées, puisque le nombre entier 7 et le nombre à virgule 7,0 sont exprimés d'une manière très différente (voir le chapitre 7). D'autre part, les types permettent d'éviter un certain nombre d'erreurs : par exemple l'instruction :

```
if (3 + 4) {
  System.out.println(x);}
else {
  System.out.println(y);}
```

contient une erreur puisque l'expression attendue après le `if` doit être de type `boolean` et non de type `int`. Les types jouent ici un rôle comparable à celui des dimensions en physique où l'équation $F = m\,g^2$ est non seulement fausse, mais de plus mal formée, car le membre de gauche est exprimé en $kg\ m\ s^{-2}$ alors que celui de droite est exprimé en $kg\ m^2\ s^{-4}$.

On peut changer le type d'une expression en la préfixant par le nom d'un type, par exemple, si la valeur de l'expression `e` est le nombre entier 7, alors celle de l'expression `(double) e` est le nombre à virgule 7,0. Si la valeur de l'expression `e` est le nombre à virgule 4,0, alors celle de l'expression `(int) e` est le nombre entier 4.

Si la valeur de l'expression `e` est un nombre à virgule, comme 3,4, qui ne correspond pas à un nombre entier, alors la valeur de l'expression `(int) e` est la partie entière de la valeur de `e`, dans cet exemple 3. Attention, cette partie entière est inhabituelle pour les nombres négatifs : si la valeur de l'expression `e` est -3,4, alors celle de `(int) e` est -3 et non -4. Il vaut donc mieux systématiquement utiliser d'abord la fonction `Math.floor` qui, à chaque nombre à virgule x, associe un autre nombre à virgule qui est la partie entière de x, avant de changer le type d'une expression. Ainsi, si la valeur de de l'expression `e` est -3,4, alors celle de l'expression `Math.floor(e)` est -4,0 et celle de l'expression `(int) Math.floor(e)` est le nombre entier -4.

Savoir-faire Différencier les types de base

Plusieurs facteurs peuvent déterminer le type à donner à une variable.

- Que représente-t-elle ?
- Au cours de l'exécution de l'algorithme, quelles valeurs peut-elle prendre ?
- Quelles opérations réalise-t-on avec cette variable ?

Exercice 3.3 (avec corrigé)

Quel est le type approprié pour les variables suivantes :

1. une variable qui contient le prénom de l'utilisateur,
2. une variable qui sert de compteur dans une boucle `for`,
3. une variable qui stocke au fur et à mesure le plus grand des nombres qu'un utilisateur tape au clavier,
4. une variable qui sert à calculer le nombre d'atomes dans l'Univers,
5. une variable qui permet de traiter différemment un nombre selon qu'il est pair ou impair.

1. *On cherche ici à stocker une suite de lettres : c'est une variable de type* `String`.
2. *Le corps de la boucle est évidemment exécutée un nombre entier de fois : un compteur de boucle est toujours de type* `int`.
3. *On ne sait pas a priori quels types de nombres seront tapés par l'utilisateur et on ne le maîtrise pas : il vaut donc mieux prévoir une variable de type* `double`. *Cependant, si le programme est conçu de manière à n'accepter que des nombres entiers en entrées, on donne le type* `int` *à cette variable.*
4. *À première vue on parle ici d'un nombre entier, mais le type* `int` *ne conviendra pas : sa valeur maximale est 2147483647 et il y a de l'ordre de 10^{80} atomes dans l'Univers. Il faut donc utiliser plutôt une variable de type* `double`. *Le nombre calculé sera peu précis, mais il s'agit, de toute façon, d'une estimation.*
5. *Une variable de type* `boolean` *: on ne s'intéresse pas ici au nombre lui-même mais uniquement à sa parité. On prendra donc par exemple* `true` *pour « pair » et* `false` *pour « impair ».*

Savoir-faire Changer le type d'une expression

Les seuls types pour lesquels la conversion a réellement un sens sont `int` et `double`.

- Certaines opérations, par exemple le sinus `Math.sin`, n'acceptent que des nombres de type `double`.
- Certaines opérations ont une signification différente selon le type de leurs arguments, par exemple la division de deux entiers est la division euclidienne, alors que la division de deux nombres à virgules est la division décimale.

Exercice 3.4 (avec corrigé)

Quelle est la valeur de l'expression `((double) 5 / (double) 2)` ? Et celle de l'expression `(double) (5 / 2)` ?

Dans le premier cas les nombres à diviser sont convertis en nombres à virgule avant la division : il s'agit donc de la division décimale 5,0 / 2,0, dont le résultat est 2,5.

Dans le second cas la division 5 / 2 est une division euclidienne, et son résultat est donc 2. Ce nombre est ensuite converti en un nombre à virgule ce qui donne le nombre 2,0.

Exercice 3.5

En utilisant la fonction `Math.random`, écrire un programme qui génère de façon équiprobable des nombres entiers entre 1 et *n*, où *n* est un entier donné.

Exercice 3.6

Modifier le programme de bataille navale pour que la position du bateau soit choisie au hasard à chaque exécution.

Exercice 3.7 (avec corrigé)

Arrondir la valeur de la variable `x` au millième près.

La fonction `Math.floor` *ne garde que la partie entière du nombre à virgule donné. Pour garder les trois premières décimales, il faut donc les faire passer temporairement dans la partie entière. Le nombre* `Math.floor(x * 1000.0)` *a donc les chiffres recherchés mais il est 1000 fois trop grand.*

Pour retrouver l'arrondi recherché, il faut donc effectuer sa division décimale par 1000. L'expression recherchée est donc `Math.floor(x * 1000.0) / 1000.0`*.*

Exercice 3.8 (avec corrigé)

Le programme suivant devrait permettre de trouver la mesure principale d'un angle en radians, mais il contient une erreur. Identifier cette erreur et proposer une correction.

```
double alpha,principale;
alpha = Isn.readDouble();
principale = alpha % (2 * Math.PI);
if (principale > Math.PI) {
  principale = principale - 2 * Math.PI;}
System.out.println(principale);
```

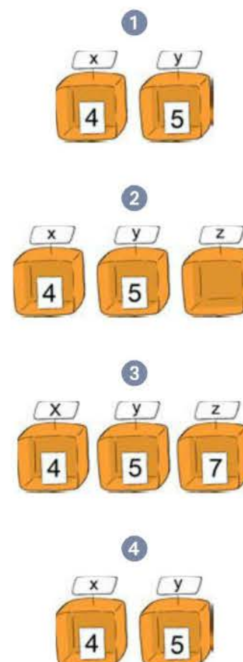
L'erreur est ici d'utiliser l'opération % qui est le reste de la division euclidienne avec des nombres de type `double`*. On est ici obligé de simuler le calcul du reste en utilisant une partie entière : si le quotient $\alpha / 2\pi$ a pour partie entière n, alors $\alpha - 2n\pi$ correspond au « reste » que l'on cherche.*

```
double alpha,principale,n;
alpha = Isn.readDouble();
n = Math.floor(alpha / (2 * Math.PI));
principale = alpha - 2 * n * Math.PI;
if (principale > Math.PI) {
  principale = principale - 2 * Math.PI;}
System.out.println(principale);
```

La portée et l'initialisation des variables

Si l'on exécute l'instruction :

```
int z;
{
  z = 7;
  System.out.println(z);}
```



dans l'état 1 on commence par ajouter une boîte z à l'état 2 puis on met la valeur 7 dans cette boîte (3) on affiche son contenu : 7 et on supprime la boîte z (4). En revanche, si dans ce même état 1 on exécute l'instruction :

```
z = 7;
System.out.println(z);
```

on déclenche une erreur : au moment où l'on cherche à remplir la boîte z, on s'aperçoit qu'il n'y a pas de boîte z dans l'état. En fait, cette erreur est même détectée avant l'exécution du programme, puisque l'on peut voir simplement en observant le programme que la variable z est utilisée, alors qu'elle n'est pas déclarée.

Une erreur similaire se produit si l'on cherche à exécuter, dans ce même état, l'instruction :

```
{
  int z;
  z = 7;}
System.out.println(z);
```

Cette instruction est, en effet, une séquence formée de l'instruction :

```
{
  int z;
  z = 7;}
```

et de l'instruction :

```
System.out.println(z);
```

L'exécution de la première de ces instructions ajoute une boîte z à l'état, remplit cette boîte avec la valeur 7 et supprime cette boîte de l'état. On se retrouve alors dans l'état initial 1 pour exécuter la seconde instruction qui tente d'afficher le contenu de la boîte z et échoue, puisqu'il n'y a plus

de boîte z dans l'état à ce moment. Dans une instruction de la forme int z; p, on ne peut utiliser la variable z que dans l'instruction p : avant la boîte n'a pas encore été ajoutée, après elle a déjà été supprimée.

Enfin, si on omet les accolades, l'instruction :

```
int z;
z = 7;
System.out.println(z);
```

est, par convention, une autre notation pour :

```
int z;
{
  z = 7;
  System.out.println(z);}
```

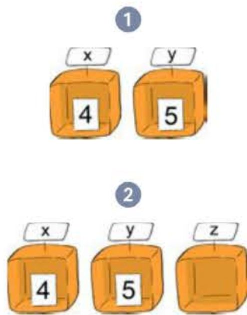
Portée d'une variable

La *portée* d'une variable z est l'ensemble des instructions au cours de l'exécution desquelles la boîte z est présente dans l'état.

qui s'exécute donc, comme ci-dessus. De manière plus générale, une instruction de la forme T x; p q est une autre notation pour T x; {p q} et non pour {T x; p} q.

Si maintenant, on cherche à exécuter l'instruction :

```
int z;
System.out.println(z);
```



dans l'état ① on commence par ajouter une boîte z à l'état ce qui donne l'état ② puis on cherche à afficher le contenu de cette boîte, mais elle est vide. L'erreur n'est pas ici due au fait que la boîte n'existe pas, mais au fait qu'elle est vide : la variable z a bien été déclarée, mais elle n'a pas été affectée. Contrairement au cas précédent, l'exécution d'un tel programme varie grandement d'un langage et d'un compilateur (voir le chapitre 15) à l'autre. Dans certains cas, une erreur se produit au moment où l'on exécute le programme : au moment d'exécuter l'instruction System.out.println(z);, on se rend compte que la boîte est vide. Dans d'autres cas, l'erreur est détectée avant l'exécution du programme : une analyse, plus difficile que la simple analyse de portée, permet de vérifier que toutes les variables sont initialisées avant d'être utilisées. Dans d'autres cas encore, aucune erreur ne se produit : la déclaration de la variable remplit la boîte z ou bien avec une valeur par défaut, en général 0, ou bien avec une valeur aléatoire.

SAVOIR-FAIRE Déclarer les variables avec des types et des portées appropriés

Le choix des types des variables est toujours dicté par les mêmes raisons (voir le savoir-faire, page 39, « Différencier les types de base »). La déclaration des variables doit donner à chaque variable une portée suffisante pour englober tous les endroits où elle est utilisée.

Exercice 3.9 (avec corrigé)

Compléter le programme suivant afin que chaque variable soit correctement déclarée.

```
n = Isn.readInt();
epsilon = Math.pow(10,-n);
racine = 1.0;
racineprec = 2.0;
while (Math.abs(racine - racineprec) > epsilon) {
  racineprec = racine;
  racine = 1.0 / (2.0 + racineprec);}
System.out.println(racine + 1.0);
```

On dénombre quatre variables dans ce programme.

- *L'affectation initiale de* `n` *impose directement qu'elle soit de type* `int`*.*
- *La fonction* `Math.pow` *renvoie un* `double`*, c'est donc le type de* `epsilon`*, qui de plus sera égal à 10^{-n}.*
- *Les variables* `racine` *et* `racineprec` *sont initialisées par des valeurs de type* `double`*, elles sont donc de type* `double`*.*

Aucune variable n'est strictement locale à la boucle `while`*. On ajoutera donc toutes les déclarations correspondantes en début de programme, juste avant l'instruction* `n = Isn.readInt();`*.*

Ce programme calcule une valeur approchée à 10^{-n} près de la racine carrée de 2.

SAVOIR-FAIRE Initialiser les variables

Identifier la première ligne du programme où une variable est utilisée, hormis les déclarations. Lorsque l'algorithme comporte des tests, il peut y avoir plusieurs telles lignes pour la même variable, en fonction du résultat du test. Vérifier que cette première ligne est toujours une initialisation. Une initialisation peut être rendue difficile à repérer parce qu'elle est à l'intérieur d'une instruction `for` ou parce qu'elle demande une saisie au clavier. Enfin, si une initialisation est manquante, il faut déterminer une valeur d'initialisation cohérente avec la suite du programme et ajouter l'instruction correspondante avant la première utilisation de la variable.

Exercice 3.10 (avec corrigé)

Quel problème peut se poser avec le programme suivant ? Le corriger.

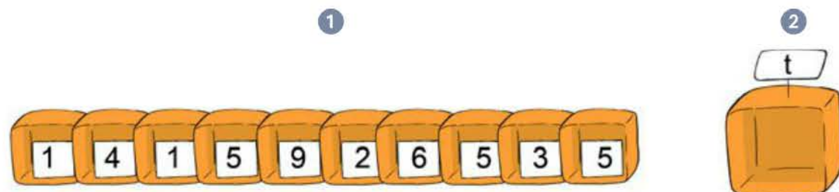
```
int n, i, f;
n = Isn.readInt();
for (i = 1; i <= n; i = i + 1) {
  f = f * i;}
System.out.println(f);
```

La variable `f`, qui sert à calculer la factorielle de `n`, n'est pas initialisée et le résultat sera donc faussé par la valeur qu'elle contient par défaut. Les opérations qui sont effectuées sur `f` sont toutes des multiplications. Pour que la valeur initiale de `f` n'ait pas d'influence sur ces calculs, il faut que ce soit 1. On ajoutera donc la ligne `f = 1;` après les déclarations des variables. Cette initialisation est d'ailleurs cohérente avec la convention selon laquelle 0 ! = 1.

Les tableaux

Jusqu'à présent, on a écrit des programmes qui utilisent les types `int`, `double` et `boolean`. Une boîte d'un tel type contient une valeur formée d'un unique nombre ou d'un unique booléen. Dans de nombreuses situations, on a besoin d'utiliser des valeurs qui, comme les textes, les images ou les sons, sont formées de plusieurs nombres ou de plusieurs booléens. Ces valeurs sont dites de type *composite*. On a commencé à voir un exemple d'un tel type, puisqu'une valeur de type `String` est une chaîne formée de plusieurs caractères. Dans cette section on va plus loin, en introduisant de nouveaux types composites : les types de tableaux.

Si on veut utiliser une boîte qui contient dix nombres entiers, par exemple, les dix premières décimales du nombre π ❶ on commence par déclarer une variable `t` de type *tableau d'entiers* : `int []`. Toutefois, déclarer une telle variable ajoute à l'état, non pas une grande boîte à dix cases comme ❶, mais une petite boîte qui a une unique case ❷.



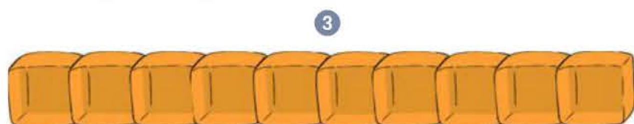
Tableau

Une boîte pouvant contenir plusieurs valeurs du même type s'appelle un *tableau*.

Pour ajouter à l'état une grande boîte, on utilise une nouvelle construction : l'*allocation* d'un tableau `new T [e]` où `T` est un type et `e` une expression de type `int`. Évaluer cette expression a pour effet

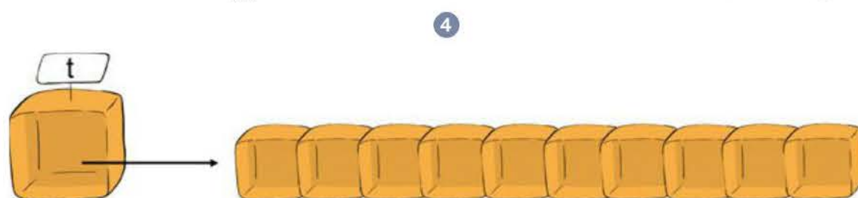
d'ajouter à l'état une boîte dont le nombre de cases est la valeur de l'expression `e`, chaque case pouvant contenir une valeur de type `T`.

Par exemple, évaluer l'expression `new int [10]` ajoute à l'état un tableau de dix cases, chaque case pouvant contenir un nombre entier (3).



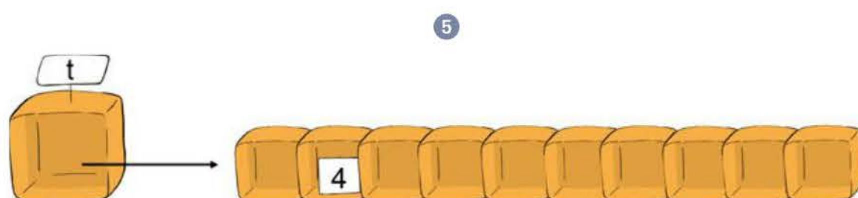
L'allocation est, avec la déclaration d'une variable, l'une des deux constructions qui permettent d'ajouter une boîte à l'état.

La valeur de l'expression `new T [e]` est la *référence* du tableau ajouté à l'état. On peut ensuite utiliser une affectation pour mettre cette valeur dans une boîte de type `int []` avec l'affectation `t = new int [10];` (4).



Avec cette notion d'allocation d'un tableau apparaissent donc les nouvelles notions de *référence* d'une boîte et de boîte contenant la référence d'une autre boîte, ce qui est symbolisée par la flèche dans le dessin ci-dessus.

Les deux dernières constructions qui permettent d'utiliser les tableaux sont l'*affectation* d'une case d'un tableau et l'*accès* à une case d'un tableau. Si `t` est le nom d'une boîte, dont le contenu est la référence d'un tableau à n cases, `e` est une expression dont la valeur est un nombre entier p compris entre 0 et $n - 1$ et `e'` une expression, alors l'exécution de l'instruction `t[e] = e';` a pour effet de remplir la case numéro p de ce tableau avec la valeur de l'expression `e'`. Par exemple, exécuter l'instruction `t[1] = 4;` produit l'état 5 et exécuter l'instruction `t[3 - 2] = 2 + 2;` produit naturellement ce même état.



Enfin, on accède à une case d'un tableau avec l'expression `t[e]`. Si `t` est le nom d'une boîte, dont le contenu est la référence d'un tableau à n cases et `e` est une expression dont la valeur est un nombre entier p compris entre 0 et $n-1$, alors la valeur de l'expression `t[e]` est la valeur contenue dans la case numéro p de ce tableau. Par exemple, la valeur de l'expression `t[1]` dans l'état ⑤ est 4.

Les quatre constructions qu'il est nécessaire de maîtriser pour utiliser les tableaux sont donc :

- la construction d'un type tableau `T []`, par exemple `int []`,
- l'allocation d'un tableau `new T [e]`, par exemple `new int [10]`,
- l'affectation d'une case d'un tableau `t[e] = e'`, par exemple `t[1] = 4;`
- l'accès à une case d'un tableau `t[e]`, par exemple `t[1]`.

SAVOIR-FAIRE Utiliser un tableau dans un programme

1 Déclarer une variable de type tableau.
2 Allouer le tableau, en lui donnant une taille n adéquate.
3 Veiller à ce que chacune des cases du tableau soit initialisée avant d'être utilisée.
4 Utiliser les affectations `t[e] = e';` et les accès `t[e]` en veillant à ce que la valeur de l'expression `e` soit bien comprise entre les bornes 0 et $n-1$.

Exercice 3.11 (avec corrigé)

Construire un répertoire associant des numéros de téléphone à des noms.

On utilise deux tableaux l'un contenant les noms et l'autre les numéros de téléphone, tels que le numéro `tel[i]` *soit le numéro de téléphone de la personne dont le nom est* `nom[i]`*. Retrouver le numéro associé au nom* `s`*, consiste à parcourir le tableau* `nom`*, jusqu'à trouver un indice* `i` *tel que* `s` *soit égal à* `nom[i]`*, puis à afficher le numéro correspondant à cet indice.*

```
String [] nom, tel;
int i;
String s;
nom = new String [10];
tel = new String [10];
// Remplissage du répertoire
nom[0] = "Alice";
tel[0] = "0606060606";
nom[1] = "Bob";
tel[1] = "0606060607";
nom[2] = "Charles";
tel[2] = "0606060608";
nom[3] = "Djamel";
tel[3] = "0606060609";
```

```
nom[4] = "Étienne";
tel[4] = "0606060610";
nom[5] = "Frédérique";
tel[5] = "0606060611";
nom[6] = "Guillaume";
tel[6] = "0606060612";
nom[7] = "Hector";
tel[7] = "0606060613";
nom[8] = "Isabelle";
tel[8] = "0606060614";
nom[9] = "Jérôme";
tel[9] = "0606060615";
// Recherche du numéro associé au nom s
s = Isn.readString();
i = 0;
while (i < 10 && !Isn.stringEqual(s,nom[i])) {
  i = i + 1;}
if (i < 10) {
  System.out.println(tel[i]);}
else {
  System.out.println("Inconnu");}
```

Exercice 3.12

Écrire un programme qui lit au clavier une chaîne de caractères et la traduit en Morse, par exemple la chaîne « sos » se traduit en « … / – – – / … » .

Exercice 3.13

Écrire un programme qui compte le nombre d'éléments supérieurs à 10 dans un tableau d'entiers.

Exercice 3.14

Écrire un programme qui trouve l'élément maximal dans un tableau d'entiers.



Exercice 3.15

On se donne un tableau d'entiers. Écrire un programme qui range les éléments de ce tableau dans un autre tableau, en mettant les éléments pairs à gauche et les éléments impairs à droite. Même exercice en utilisant un seul tableau.

Exercice 3.16

Qu'affiche le programme suivant ?

```
int [] t;
t = new int [10];
t[1] = 4;
System.out.println(t[1]);
```

Et le programme suivant ?

```
int [] t;
t[1] = 4;
System.out.println(t[1]);
```

Les tableaux bidimensionnels

❶

120	145	87
12	67	89
90	112	83

❷

120 145 87 12 67 89 90 112 83

Pour représenter une table à double entrée, par exemple la table ❶, une possibilité est d'utiliser un tableau de neuf cases (❷) mais cette manière de faire est malcommode.

On préfère donc, en général, représenter une telle table par un tableau de trois éléments dont chaque élément est la représentation d'une colonne, c'est-à-dire lui-même un tableau de trois nombres. Ainsi une table de nombre à double entrée est un tableau de tableaux de nombres : un objet de type `int [][]`.

Si `t` est une variable d'un tel type, la valeur de la case d'abscisse `x` et d'ordonnée `y`, c'est-à-dire de la case de la colonne `x` et de la ligne `y`, est simplement désignée par l'expression `t[x][y]`. Affecter une case du tableau `t` se fait simplement par l'instruction `t[x][y] = e;` où `e` est une expression de type `int`. Seule l'allocation d'un tableau se fait d'une manière particulière : en principe, il faudrait allouer le tableau `t`, puis allouer un tableau pour chacune des colonnes. Cependant, une nouvelle expression permet d'allouer tout d'un coup : `new T[e][e']` où `T` est un type et `e` et `e'` deux expressions de type `int`. Évaluer cette expression a comme effet d'ajouter à l'état un tableau bidimensionnel dont le nombre de colonnes est la valeur de l'expression `e`, et le nombre de lignes est la valeur de l'expression `e'`, chaque case pouvant contenir une valeur de type `T`. Par exemple, évaluer l'expression `new int [3][3]` ajoute à l'état un tableau de trois colonnes sur trois lignes, chaque case contenant une valeur de type `int`.



Exercice 3.17

Écrire un programme qui, à l'aide de la formule de Pascal, calcule les premiers coefficients binomiaux.

1. On lira au clavier un entier `n`, puis on remplira un tableau bidimensionnel de `double coeffs_bin` de sorte que la case `coeffs_bin[i][j]` contienne le coefficient binomial $\binom{i}{j}$ pour tout $i < n$ et pour tout $j \leq i$. Les cases pour lesquelles $j > i$ contiendront la valeur 0.
2. À partir de quelle valeur de `n` les nombres calculés dépassent-ils la capacité de la mantisse du type `double` ?
   Les coefficients binomiaux calculés pour une valeur de `n` supérieure à celle-ci sont-ils exacts ? Sinon, de quel ordre est l'approximation réalisée ?
3. À partir de quelle valeur de `n` les nombres calculés dépassent-ils la capacité du type `double` ?
   Que peut-on penser des coefficients binomiaux calculés pour une valeur de `n` supérieure à cette dernière ?

ALLER PLUS LOIN Qu'est-ce que le calcul formel ?

Comment calculer avec la fonction polynôme $x \mapsto 2\,x^3 + 8\,x^2 + 7\,x + 3$? Une possibilité est d'en faire une expression de son programme. Cela permet, par exemple, de calculer sa valeur en 5.

```
int x, y;
x = 5;
y = 2 * x * x * x + 8 * x * x + 7 * x + 3;
System.out.println(y);
```

ce qui donne, bien entendu, le résultat 488.

Mais, il est aussi possible de représenter une fonction polynôme du troisième degré par le quadruplet de ses coefficients, c'est-à-dire par un tableau

```
double [] t;
```

On peut alors définir la fonction polynôme $x \mapsto 2\,x^3 + 8\,x^2 + 7\,x + 3$ ainsi

```
t = new double [4];
t[3] = 2;
t[2] = 8;
t[1] = 7;
t[0] = 3;
```

Calculer la valeur de cette fonction en 5 demande un programme un peu plus complexe

```
int x, y, i, c;
x = 5;
y = 0;
c = 1;
for (i = 0; i <= 3; i = i + 1) {
  y = y + t[i] * c;
  c = c * x;}
System.out.println(y);
```

où la variable `c` contient les valeurs des puissances successives de `x`.

Mais représenter cette fonction ainsi permet de faire de nombreuses nouvelles choses, par exemple l'afficher

```
for (i = 0; i <= 3; i = i + 1) {
  System.out.print(t[i]);
  System.out.print(" ");
  if (i != 0) {
    System.out.print("x");
    if (i != 1) {
      System.out.print("^");
      System.out.print(i);}}
  if (i != 3) {
    System.out.print(" + ");}}
System.out.println();
```

ce qui donne le résultat

```
3.0 + 7.0 x + 8.0 x^2 + 2.0 x^3
```

et même calculer sa dérivée, en utilisant le fait que la dérivée de x^{n+1} est $(n + 1)\,x^n$

```
for (i = 0; i <= 2; i = i + 1) {
  u[i] = t[i+1] * (i + 1);}
u[3] = 0;
```

que l'on peut à son tour afficher

```
7.0 + 16.0 x + 6.0 x^2 + 0.0 x^3
```

et on obtient que la fonction $x \mapsto 6\,x^2 + 16\,x + 7$ est la dérivée de la fonction $x \mapsto 2\,x^3 + 8\,x^2 + 7\,x + 3$. Cette procédure est bien sûr valable quelle que soit la fonction polynôme du troisième degré représentée dans le tableau `t`.

Les programmes peuvent donc calculer avec des objets très divers : des nombres et des chaînes de caractères bien entendu, mais aussi des expressions symboliques comme $x \mapsto x^3 + 8\,x^2 + 7\,x + 3$.



Exercice 3.18

On souhaite écrire un programme qui détermine les bornes de l'intervalle de fluctuation au seuil de 95 % d'une loi binomiale de paramètres `n` et `p` qu'on lira au clavier.

1. Écrire une version de ce programme qui calcule la loi de probabilité binomiale dans son intégralité puis détermine et affiche l'intervalle de confiance.
2. Écrire une version de ce programme qui détermine et affiche l'intervalle de confiance en ne calculant la loi de probabilité que jusqu'à atteindre la borne supérieure de l'intervalle de confiance. Quels calculs économise-t-on de cette façon ?
3. Si `p` est proche de 1, que peut-on dire du programme écrit à la question 2 ? Proposer une troisième version pour rendre le programme plus efficace dans ce cas particulier.

4. Comment décider s'il vaut mieux utiliser le programme écrit à la question 2 ou celui écrit à la question 3 ? Écrire un programme qui fait ce choix automatiquement.
5. En utilisant la méthode détaillée au chapitre 21, déterminer si la partie du programme qui prend le plus de temps est celle où l'on calcule les coefficients binomiaux ou bien celle où l'on détermine l'intervalle de fluctuation.
6. En déduire si les différentes versions écrites dans les questions 1 à 4 modifient significativement le temps d'exécution de l'algorithme. En admettant que l'on n'ait jamais à utiliser cet algorithme pour des valeurs de n supérieures à 1000, comment peut-on le rendre plus efficace ?

Les chaînes de caractères

Jusqu'à présent, on a défini et manipulé le type `String` comme si c'était un type de base, alors qu'il est en fait composite : une chaîne de caractères est formée de plusieurs valeurs simples que sont ses différents caractères.

SAVOIR-FAIRE Calculer avec des chaînes de caractères

- Pour comparer des chaînes de caractères, on utilise impérativement les fonctions `Isn.stringEqual` et `Isn.stringAlph`. Attention en particulier à l'opération `==` qui ne produira pas d'erreur à la compilation mais ne donnera pas les résultats attendus.
- Lorsque l'on doit parcourir une chaîne de caractères élément par élément, on peut utiliser une boucle `for`, le nombre d'exécutions du corps de la boucle étant donné par la fonction `Isn.stringLength`.
- Pour modifier une chaîne existante, on en extrait les parties appropriées à l'aide de la fonction `Isn.stringNth`, on reconstitue une nouvelle chaîne à l'aide de l'opération de concaténation `+`, et on l'affiche ou on la stocke dans une variable.

Exercice 3.19 (avec corrigé)

Écrire un programme qui demande à l'utilisateur de taper son prénom et son nom, puis affiche les initiales correspondantes.

Si le prénom et le nom sont entrés dans deux chaînes différentes, il suffit d'afficher le premier caractère de chacune de ces chaînes.

```
String prenom, nom;
prenom = Isn.readString();
nom = Isn.readString();
System.out.print(Isn.stringNth(prenom,0));
System.out.println(Isn.stringNth(nom,0));
```

Mais ce programme ne permet pas de traiter des prénoms composés ou des noms multiples. Si le prénom et le nom sont entrés dans une seule chaîne, les initiales sont les premières lettres des mots, c'est-à-dire le premier caractère de la chaîne et ceux qui suivent immédiatement un espace. Il faut donc parcourir cette chaîne caractère par caractère en recherchant les espaces, et afficher le caractère suivant dès que l'on en trouve un. On ne va en réalité que jusqu'à l'avant-dernier caractère de la chaîne, car même si le dernier caractère est un espace, il ne peut pas être suivi d'une lettre.

```
String nom;
int i;
nom = Isn.readString();
System.out.print(Isn.stringNth(nom,0));
for (i = 0; i <= Isn.stringLength(nom) - 2; i = i + 1) {
  if (Isn.stringEqual(" ",Isn.stringNth(nom, i))) {
    System.out.print(Isn.stringNth(nom,i + 1));}}
System.out.println();
```

Cette version est plus générale mais aussi plus sensible aux entrées mal formées : un espace mal placé ou un tiret utilisé pour séparer des prénoms fausse le résultat.

Exercice 3.20

Écrire un programme qui lit une chaîne de caractères et :

1. compte le nombre d'espaces,
2. compte le nombre de voyelles,
3. calcule le score marqué au Scrabble avec cette chaîne, en comptant 0 point pour les espaces et les signes de ponctuation,
4. détermine la lettre la plus fréquente.

Exercice 3.21

Écrire un programme qui lit une chaîne de caractères et la réécrit en revenant à la ligne entre chaque caractère.



Exercice 3.22

Chercher sur le Web ce qu'est la méthode de chiffrement ROT13 et écrire un programme qui lit une chaîne de caractères et la chiffre ou la déchiffre selon cette méthode.



Exercice 3.23

Écrire un programme qui lit une chaîne de caractères et :

1. la réécrit tout en majuscules,
2. la réécrit tout en minuscules,
3. la réécrit en inversant majuscules et minuscules.

Exercice 3.24

Chercher sur le Web ce qu'est l'écriture « leet speak » et écrire un programme qui lit une chaîne de caractères et la réécrit en leet speak.

La mise au point des programmes

Les programmes sont souvent des objets complexes formés de plusieurs milliers, voire plusieurs millions de lignes et il est peu probable, quand on écrit un programme qui dépasse quelques dizaines de lignes, de ne pas faire d'erreur. Une erreur dans un programme peut avoir des conséquences dramatiques si ce programme est par exemple utilisé dans le régulateur de vitesse d'une voiture, une centrale nucléaire ou un robot chirurgical. C'est pourquoi il existe de nombreuses méthodes pour éviter les erreurs dans les programmes. La première, déjà évoquée au chapitre 1, est de tester les programmes que l'on écrit.

Quand on teste un programme et qu'il ne fait pas ce que l'on espère, il faut déterminer l'endroit où une erreur s'est produite. Pour cela on doit *instrumenter* son programme, c'est-à-dire ajouter des instructions de sortie dans le programme, qui permettent de visualiser ce qu'il se passe au cours de son exécution. On repère ainsi le moment de l'exécution du programme, où une variable prend, pour la première fois, une valeur inadéquate.

SAVOIR-FAIRE Mettre au point un programme en l'instrumentant

1 Identifier les variables critiques, dont la valeur peut radicalement influencer le comportement du programme. En particulier, la variable `i` dans une instruction de la forme `for (i = e; i <= e'; i = i + 1) p` et les variables intervenant dans l'expression `e` d'une instruction de la forme `while (e) p` sont particulièrement importantes puisqu'elles conditionnent le nombre de fois que le corps de cette boucle est exécuté.

2 Identifier pour chacune de ces variables les endroits clés du programme qui la concernent : par exemple lorsque l'on lui affecte une valeur, en début ou en fin de boucle.

3 Insérer un affichage à l'écran de chaque variable critique aux endroits appropriés, en n'oubliant pas de préciser de quelle variable on affiche la valeur.

Exercice 3.25 (avec corrigé)

Le programme suivant est censé calculer la somme des carrés des n premiers entiers.

```
int i, n, somme;
n = Isn.readInt();
somme = 0;
for (i = 1; i <= n; i = i + 1) {
  i = i * i;
  somme = somme + i; }
System.out.println(somme);
```

Montrer que ce programme est erroné à l'aide d'un test bien choisi.

Instrumenter ce programme pour détecter l'erreur et la corriger.

En testant le programme pour n *= 0, 1 ou 2, on observe que le résultat affiché est correct. En revanche, pour* n *= 3, on devrait trouver 1 × 1 + 2 × 2 + 3 × 3 = 14, or le programme affiche 5.*

La variable critique est ici le compteur de boucle i*, qui sert également dans les calculs. On affichera sa valeur à la fin du corps de la boucle* for*. Pour cela on insère à cet endroit les lignes :*

```
System.out.print("i vaut ");
System.out.println(i);
```

On constate alors que i *contient 1 à la fin de la première itération, mais 4 à la fin de la deuxième itération et 25 à la fin de la troisième. Il y a donc une instruction dans le corps de la boucle qui modifie la valeur de* i*, ce qu'il ne faut pas faire dans une boucle* for*. Une fois ceci constaté, il n'est pas difficile d'incriminer la ligne* `i = i * i;`*.*

Outre le test et l'instrumentation, il existe de nombreuses autres méthodes de mise au point des programmes. Dans certains langages, les types permettent de détecter des erreurs de façon beaucoup plus poussée qu'en Java. On peut aussi démontrer qu'un programme vérifie certaines propriétés (voir le chapitre 18), ce qui évite en particulier d'avoir à chercher les erreurs à tâtons.

ALLER PLUS LOIN

Pour de gros programmes

Une telle méthode est utilisable sur des programmes de petite taille, mais peut rapidement devenir lourde à mettre en place pour de gros programmes : il faut décider des variables à observer et des points du programme où cela a un intérêt, puis modifier le programme et le recompiler, et enfin supprimer les affichages du programme lorsque l'on a identifié le problème. Dans de nombreux environnements de développement logiciel, il existe des outils spécialisés, les *débogueurs*, pour obtenir des informations sur ce qu'il se passe en mémoire durant l'exécution d'un programme, ou bien exécuter un programme pas à pas.

Aller plus loin Un langage de programmation petit, mais complet

Maintenant que nous avons introduit la notion de déclaration, nous pouvons construire un petit de langage de programmation qui contient

- la déclaration de variables,
- l'affectation,
- la séquence,
- le test
- et la boucle `while`.

Ce langage de programmation, bien qu'il soit très petit, est *complet*, ce qui signifie que tous les programmes que l'on peut imaginer peuvent être exprimés dans ce langage.

Nous avons vu que l'instruction :

```
for (i = e; i <= e'; i = i + 1) p
```

pouvait être traduite en l'instruction :

```
i = e;
while (i <= e') {p i = i + 1;}
```

qui ne contient que des affectations, des séquences et une boucle `while`. La boucle `for` n'est donc qu'une manière plus confortable d'écrire des programmes qui pourraient s'exprimer dans ce petit langage. De même, les instructions qui utilisent les fonctions et la récursivité que nous introduirons aux chapitres 4 et 5, pourraient, en théorie, être traduites dans ce petit langage, même si ces traductions sont beaucoup plus complexes que celle de la boucle `for`.

De même que tous les objets qui nous entourent sont formés de trois types de particules : les protons, les neutrons et les électrons, que tous les textes que nous lisons sont formés de vingt-six lettres, et de quelques signes de ponctuation, que toutes les pièces de musiques que nous entendons sont formées de douze notes, tous les programmes que nous utilisons peuvent ultimement être exprimés avec ces cinq instructions : la déclaration, l'affectation, la séquence, le test et la boucle.

Ai-je bien compris ?

- Comment la déclaration d'une variable transforme-t-elle l'état de l'exécution d'un programme ?
- Quels sont les différents types possibles pour une variable ?
- Qu'est-ce que la portée d'une variable ?

Les fonctions

4

Chapitre avancé

Pour avoir du style, il faut éviter les redites.

Dans ce chapitre, nous introduisons une nouvelle construction : la définition de fonction, qui permet d'isoler une instruction qui revient plusieurs fois dans un programme.

Une fonction est définie par un nom, par ses arguments qui porteront les valeurs communiquées par le programme principal à la fonction au moment de son appel et éventuellement une valeur de retour communiquée au programme par la fonction en fin d'exécution. Nous revenons dans ce chapitre sur la question de la portée des variables dans le cas des programmes qui comportent des fonctions. Nous introduisons aussi des variables *globales* dont la portée est le programme tout entier.



John McCarthy (1927 - 2011) est l'auteur du langage Lisp (1958), dont la principale construction est la définition de fonctions. Il est aussi l'un des inventeurs de la notion de temps partagé, qui permet à plusieurs personnes d'utiliser un même ordinateur en même temps. Il a écrit l'un des premiers programmes jouant aux échecs et inventé pour cela un algorithme, *la méthode alpha-bêta*, qui permet de jouer non seulement aux échecs, mais aussi à de nombreux autres jeux. Il a aussi été un défenseur de l'idée de progrès et de l'importance des mathématiques dans l'éducation.

Isoler une instruction

Au cours des chapitres précédents, il nous est arrivé d'écrire des programmes dans lesquels certaines instructions revenaient plusieurs fois, parce que nous voulions faire plusieurs fois la même chose. Un exemple de programme présentant des répétitions est le suivant :

```
System.out.print("Le vol en direction de ");
System.out.print("Tokyo");
System.out.print(" décollera à ");
System.out.print("9h00");
System.out.println();
System.out.println("----------------------------------------");
System.out.println();

System.out.print("Le vol en direction de ");
System.out.print("Sydney");
System.out.print(" décollera à ");
System.out.print("9h30");
System.out.println();
System.out.println("----------------------------------------");
System.out.println();

System.out.print("Le vol en direction de ");
System.out.print("Toulouse");
System.out.print(" décollera à ");
System.out.print("9h45");
System.out.println();
System.out.println("----------------------------------------");
System.out.println();
```

dans lequel l'instruction

```
System.out.println();
System.out.println("----------------------------------------");
System.out.println();
```

est répétée trois fois. Au lieu de répéter la totalité de cette instruction, on peut lui donner un nom, par exemple `tirerUnTrait`, puis la remplacer par ce nom dans le programme principal à chaque fois qu'elle est utilisée.

Fonction

Dans les langages de programmation, une *fonction* est une instruction isolée du reste du programme, qui possède un nom, et qui peut être appelée par ce nom à n'importe quel endroit du programme et autant de fois que l'on veut.

Pour définir la fonction `tirerUnTrait`, on procède de la façon suivante :

```
static void tirerUnTrait () {
  System.out.println();
  System.out.println("----------------------------------------");
  System.out.println();}
```

et on utilise ensuite cette fonction dans le programme principal, comme si c'était une instruction du langage :

```
System.out.print("Le vol en direction de ");
System.out.print("Tokyo");
System.out.print(" décollera à ");
System.out.print("9h00");
tirerUnTrait();

System.out.print("Le vol en direction de ");
System.out.print("Sydney");
System.out.print(" décollera à ");
System.out.print("9h30");
tirerUnTrait();

System.out.print("Le vol en direction de ");
System.out.print("Toulouse");
System.out.print(" décollera à ");
System.out.print("9h45");
tirerUnTrait();
```

Cette définition de fonction se place avant le programme principal, si bien que l'organisation générale du programme est la suivante :

```
class Horaire {

  static void tirerUnTrait () {
    System.out.println();
    System.out.println("-----------------------------------");
    System.out.println();}

  public static void main (String [] args) {

    System.out.print("Le vol en direction de ");
    System.out.print("Tokyo");
    System.out.print(" décollera à ");
    System.out.print("9h00");
    tirerUnTrait();

    System.out.print("Le vol en direction de ");
    System.out.print("Sydney");
    System.out.print(" décollera à ");
    System.out.print("9h30");
    tirerUnTrait();

    System.out.print("Le vol en direction de ");
    System.out.print("Toulouse");
    System.out.print(" décollera à ");
    System.out.print("9h45");
    tirerUnTrait();}}
```

Corps d'une fonction

L'instruction

```
System.out.println();
System.out.println("----...----");
System.out.println();
```

que l'on isole par la définition de la fonction `tirerUnTrait` s'appelle le *corps* de cette fonction.

Appel d'une fonction

L'instruction `tirerUnTrait();` s'appelle un *appel* de la fonction tirerUnTrait. Exécuter cette instruction a pour effet d'exécuter le *corps* de la fonction.

Utiliser des fonctions évite les répétitions dans les programmes et rend donc ces derniers plus courts et, surtout, plus faciles à lire et à comprendre : pour comprendre le programme ci-dessus, il n'est pas nécessaire de savoir comment la fonction `tirerUnTrait` est définie, il suffit de savoir ce qu'elle fait. Utiliser des fonctions permet aussi d'organiser le

travail de développement : on peut décider d'écrire le programme principal un jour et d'écrire la fonction `tirerUnTrait` le lendemain. On peut aussi décider de confier l'écriture du programme principal à un programmeur et l'écriture de la fonction `tirerUnTrait` à un autre. Enfin, si l'on veut modifier la longueur du trait à tirer ou le nombre de lignes sautées au-dessus et en-dessous de ce trait, il suffit de modifier le corps de la fonction et non le programme principal à chacun des endroits concernés.

Passer des arguments

Le programme ci-précédent est formé de trois blocs qui annoncent chacun l'horaire d'un vol. On peut vouloir aller plus loin dans l'organisation de ce programme et écrire une fonction `annoncerUnVol`, qu'il suffirait d'appeler trois fois dans le programme principal. Cependant, contrairement à l'exemple de la fonction `tirerUnTrait`, ces trois blocs ne sont pas absolument identiques : la destination et l'horaire du vol diffèrent d'un cas à l'autre. Il faut donc *paramétrer* l'instruction que l'on isole pour pouvoir choisir la destination et l'horaire du vol.

Argument d'une fonction

On appelle *argument* d'une fonction une variable particulière, utilisée dans le corps de la fonction, et dont la valeur est donnée dans le programme principal au moment où la fonction est appelée.

Dans notre exemple, les arguments doivent représenter la destination, que l'on nomme `destination`, et l'horaire de vol, que l'on nomme `horaire`. On définit alors cette fonction de la manière suivante :

```
static void annoncerUnVol (String destination, String horaire) {
  System.out.print("Le vol en direction de ");
  System.out.print(destination);
  System.out.print(" décollera à ");
  System.out.print(horaire);
  System.out.println();
  System.out.println("--------------------------------------------");
  System.out.println();}
```

Et le programme principal devient :

```
annoncerUnVol("Tokyo","9h00");
annoncerUnVol("Sydney","9h30");
annoncerUnVol("Toulouse","9h45");
```

Exécuter une instruction de la forme `annoncerUnVol(e,e');` a pour effet d'évaluer les deux expressions `e` et `e'` et de fabriquer un nouvel état dans lequel deux boîtes de noms `destination` et `horaire` contiennent respectivement la valeur de l'expression `e` et celle de l'expression `e'`, d'exécuter le corps de la fonction dans cet état, puis de revenir à l'état initial. Dans

l'exemple ci-avant, au moment où l'on exécute pour la première fois l'instruction `System.out.print(destination);` du corps de la fonction, la valeur de l'expression `destination` est `"Tokyo"`. Lors du deuxième appel, la valeur de l'expression `destination` est `"Sydney"`. Et lors du troisième, elle est `"Toulouse"`.

Récupérer une valeur

Le passage d'arguments permet donc de communiquer des informations depuis le programme principal vers une fonction. On veut aussi souvent communiquer des informations dans l'autre sens : depuis une fonction, vers le programme principal. Par exemple, si l'on veut isoler, dans une fonction, l'instruction suivante qui calcule le nombre `n` de fois que le caractère `a` apparaît dans une chaîne `s` :

```
int i, n;
n = 0;
for (i = 0; i <= Isn.stringLength(s) - 1; i = i + 1) {
  if (Isn.stringEqual(Isn.stringNth(s,i), "a")) {
    n = n + 1;}}
```

On veut non seulement que le programme principal puisse communiquer la chaîne de caractères `s` à la fonction, mais aussi que la fonction puisse communiquer le nombre `n` au programme principal. Cela mène à écrire la fonction suivante :

```
static int nombreDea (String s) {
  int i, n;
  n = 0;
  for (i = 0; i <= Isn.stringLength(s) - 1; i = i + 1) {
    if (Isn.stringEqual(Isn.stringNth(s,i), "a")) {
      n = n + 1;}}
  return n;}
```

L'exécution de l'instruction `return n;` a pour effet d'interrompre l'exécution du corps de la fonction et de renvoyer la valeur de l'expression `n` au programme principal.

Comme la fonction `nombreDea` renvoie une valeur, l'appel `nombreDea("abracadabra")`, dans le programme principal, n'est pas une instruction, mais une expression, qui a la valeur 5. On peut l'utiliser, par exemple, dans une affectation `x = nombreDea("abracadabra");`.

Valeur de retour

La valeur produite par une fonction à partir de ses arguments, s'il en existe une, est appelée *valeur de retour*.

En-tête d'une fonction

On appelle *en-tête* d'une fonction la première ligne de sa définition, qui comporte dans l'ordre :

- le mot-clé `static`,
- le type de la valeur de retour de la fonction, ou `void` le cas échéant,
- le nom de la fonction,
- la liste de ses arguments entre parenthèses, chaque argument étant précédé de son type.

Si une fonction ne renvoie pas de valeur, par exemple si elle ne fait qu'afficher des messages à l'écran, on fait précéder sa définition du mot-clé `void`, sinon, on la fait précéder du type de sa valeur de retour.

Dans l'*en-tête* `static void annoncerUnVol (String destination, String horaire)` on trouve le nom `annoncerUnVol` et les deux arguments `destination` et `horaire`, l'un et l'autre de type `String`. Enfin, le mot-clé `void` indique que cette fonction ne renvoie pas de valeur.

Savoir-faire Écrire l'en-tête d'une fonction

1 Choisir un nom qui indique clairement ce que fait la fonction.
2 Identifier les arguments qui varient lors des différents appels de la fonction dans le programme principal. Donner un nom à chacun de ces arguments.
3 Identifier un type approprié pour chacun de ces arguments.
4 Identifier si la fonction renvoie une valeur et, si oui, le type de cette valeur.

Exercice 4.1 (avec corrigé)

Écrire l'en-tête d'une fonction qui calcule la vitesse moyenne d'un mobile connaissant son temps de parcours et la distance parcourue.

1. *La fonction peut, par exemple, s'appeler* `vitesse`*.*
2. *Les arguments sont tout indiqués : on les appelle, par exemple,* `temps` *et* `distance`*.*
3. *Selon les unités choisies, les arguments pourraient être de type* `int` *ou* `double`*. Cependant, on sait qu'il faut effectuer une division décimale pour calculer la vitesse et que cette opération requiert des nombres de type* `double`*.*
4. *Enfin, la fonction doit renvoyer une valeur, la vitesse, qui est elle aussi de type* `double` *à cause de la division.*

   L'en-tête de la fonction est donc `static double vitesse (double temps, double distance)`*.*

Aller plus loin L'ordre des arguments

L'ordre des arguments n'a pas d'importance pour la définition de la fonction : les en-têtes `static void annoncerUnVol (String destination, String horaire)` et `static void annoncerUnVol (String horaire, String destination)` permettent de définir la même fonction.

En revanche, lors d'un appel à cette fonction, l'ordre des arguments doit être respecté : `annoncerUnVol("Tokyo","9h00");` est correct vis-à-vis du premier en-tête, mais `annoncerUnVol("9h00","Tokyo");` n'est correct que vis-à-vis du second.

Exercice 4.2

Écrire l'en-tête des fonctions suivantes :

1. Une fonction qui indique s'il est possible de construire un triangle avec trois segments de mesures données.
2. Une fonction qui calcule le plus grand diviseur commun (PGCD) de deux nombres entiers.
3. Une fonction qui trace à l'écran un segment entre deux points.
4. Une fonction qui écrit à l'écran les initiales d'une personne dont on donne le nom complet.

SAVOIR-FAIRE Écrire une fonction

1 Écrire l'en-tête de la fonction.
2 Écrire le corps de la fonction comme si les arguments étaient déjà remplis par des valeurs.
3 Ne pas oublier l'instruction `return`, le cas échéant.
4 Prévoir une exécution correcte de la fonction quelle que soit la valeur donnée à chacun des arguments, y compris dans des cas que l'on n'a pas forcément anticipés dans le cours normal du programme principal.

Exercice 4.3 (avec corrigé)

Écrire une fonction qui effectue la division décimale de deux nombres entiers.

Définition d'une fonction

Au bout du compte, la définition d'une fonction est formée de son en-tête, puis du corps de la fonction, entre accolades.

1 *On appelle la fonction* `divisionDecimale`*. Il est spécifié ici que les arguments doivent être de type* `int` *; on va les appeler* `dividende` *et* `diviseur`*. La fonction renvoie une valeur qui ne peut être que de type* `double` *car c'est le résultat d'une division décimale. Son en-tête est donc :*
`static double divisionDecimale (int dividende, int diviseur)`.

2 *Il faut prévoir une variable de type* `double` *pour noter le quotient de la division et le renvoyer. En outre, l'intérêt de définir une telle fonction est qu'elle prend systématiquement en charge la conversion de type de ses arguments, de façon transparente pour l'utilisateur. Le corps de la fonction doit donc comporter notamment les lignes suivantes :*

```
double quotient;
quotient = ((double) dividende) / ((double) diviseur);
```

3 *On termine cette fonction par l'instruction* `return quotient;`.

4 *Si l'on veut une fonction qui prévoit tous les cas, notamment lorsque le diviseur fourni est nul, il faut également renvoyer une valeur, choisie arbitrairement mais de type conforme à celui prévu dans l'en-tête. La fonction peut donc se présenter ainsi :*

```
static double divisionDecimale (int dividende, int diviseur) {
  double quotient;
  if (diviseur == 0) {
    quotient = Double.POSITIVE_INFINITY;}
  else {
    quotient = ((double) dividende) / ((double) diviseur);}
  return quotient;}
```

Le choix est fait ici de renvoyer le `double` *spécial $+\infty$ (voir le chapitre 7) si le diviseur est nul, par analogie avec la limite de la fonction inverse lorsque* `x` *tend vers* `0` *par valeurs positives. Ce choix est discutable, d'une part car une division par zéro ne se produit pas forcément lorsque l'on recherche une limite, d'autre part car $-\infty$ est une autre limite possible pour un quotient dont le dénominateur tend vers* `0`*, par exemple la fonction inverse lorsque* `x` *tend vers* `0` *par valeurs négatives.*

Cette fonction doit donc être accompagnée d'une documentation qui précise ce qui se passe dans ce genre de cas. On aurait pu également faire le choix d'afficher un message, par exemple `System.out.println("Erreur : division par 0 interdite.");` *mais ce dernier risquerait d'interférer avec les autres sorties du programme, et si la fonction est appelée plusieurs fois au cours du programme, il ne serait pas directement possible de savoir à quel appel l'erreur s'est produite.*

Exercice 4.4

Écrire les fonctions suivantes.

1. Une fonction qui renvoie la plus grande de deux valeurs de type `int`.
2. Une fonction qui répète un même mot un certain nombre de fois au choix.
3. Une fonction, construite à partir de la fonction `Math.random`, qui tire au sort un nombre entier entre deux bornes données en arguments.
4. Une fonction qui décide s'il est possible de construire un triangle avec trois segments de mesures données.

Exercice 4.5

Écrire une fonction qui prend en argument une chaîne de caractères `s` et deux entiers `i` et `j`, et renvoie la sous-chaîne de `s` comprise entre le caractère numéro `i` inclus et le caractère numéro `j` exclu.

Le programme principal

En Java, il a été choisi de faire du programme principal une fonction particulière qui porte un nom spécial : `main`.

La portée des variables et les variables globales

Isoler l'instruction `x = 0;` dans le programme suivant :

```
public static void main (String [] args) {
  int x;
  x = 3;
  x = 5;
  x = 0;
  x = 7;
  x = 0;
  x = 4;}
```

mènerait à écrire une fonction :

```
static void reinitialise () {
  x = 0;}
```

et le programme principal :

```
public static void main (String [] args) {
  int x;
  x = 3;
  x = 5;
  reinitialise();
  x = 7;
  reinitialise();
  x = 4;}
```

Cependant, ce programme n'est pas correct. En effet, si l'état dans lequel on exécute l'instruction `reinitialise();` dans le programme principal est formé d'une boîte de nom `x` qui contient, par exemple, la valeur 5, l'état dans lequel le corps de cette fonction est exécuté ne contient en revanche aucune boîte, si bien que quand on exécute l'instruction `x = 0;` il n'y a pas de boîte de nom `x`.

Une autre manière de voir le problème est qu'en déplaçant l'instruction `x = 0;` du programme principal vers le corps de la fonction, on a sorti l'affectation de la variable `x` de la portée de cette variable. Or, la variable `x` ne peut être utilisée que dans la partie en gras du programme :

```
static void reinitialise () {
  x = 0;}

public static void main (String [] args) {
  int x;
  x = 3;
  x = 5;
  reinitialise();
  x = 7;
  reinitialise();
  x = 4;}
```

Une manière de résoudre ce problème est de déclarer la variable `x` de manière à ce que sa portée soit le programme entier.

```
static int x;

static void reinitialise () {
  x = 0;}

public static void main (String [] args) {
  x = 3;
  x = 5;
```

Variable globale ou locale

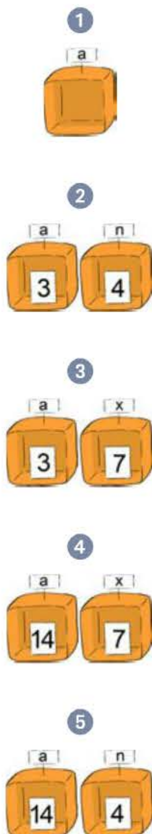
Une *variable globale* est une variable déclarée en début de programme, en dehors du programme principal et de toute fonction. Sa portée est le programme entier. En Java, la déclaration d'une variable globale est précédée du mot-clé `static`. Par opposition, une *variable locale* est déclarée à l'intérieur d'une fonction ou du programme principal, qui est une fonction particulière. Elle n'est pas utilisable depuis les autres fonctions.

```
  reinitialise();
  x = 7;
  reinitialise();
  x = 4;}
```

Ainsi, dans cet exemple, le corps de chaque fonction est exécuté dans un environnement qui contient une boîte x.

Dans un programme qui, comme celui-ci, contient des variables globales, l'appel d'une fonction a comme effet l'exécution du corps de la fonction, dans un état qui contient d'une part des boîtes qui ont comme noms les arguments de la fonction, d'autre part des boîtes qui ont comme noms les variables globales. Quand l'exécution du corps de la fonction est achevée, on supprime les boîtes qui correspondent aux arguments de la fonction, on garde celles qui correspondent aux variables globales et on remet celles qui correspondent aux variables du programme principal.

Par exemple, dans le programme suivant :




```
static int a;

static void f (int x) {
  System.out.println(2 * x);
  a = 2 * x;}

public static void main (String [] args) {
  int n;
  a = 3;
  n = 4;
  f(a + n);}
```

1 On exécute le programme principal dans l'état ❶ qui contient la variable globale a.
2 Au cours de l'exécution du programme principal, on déclare une autre variable n. On affecte la valeur 3 à la variable a et la valeur 4 à la variable n (❷).
3 Au moment de l'appel f(a + n); de la fonction f, on supprime la boîte de nom n de l'état, car n est une variable locale au programme principal, mais on garde la boîte de nom a car a est une variable globale, et on ajoute une boîte de nom x, qui contient la valeur 7 de l'expression a + n (❸).
4 On exécute alors le corps de la fonction, ce qui a pour effet d'afficher 14 et d'affecter cette valeur à la variable a (❹).
5 En quittant la fonction pour revenir au programme principal, on supprime la boîte de nom x et on remet la boîte de nom n avec le contenu qu'elle possédait avant l'appel de la fonction (❺).

SAVOIR-FAIRE Identifier la portée des variables dans un programme comportant des fonctions

- Si une variable est déclarée en dehors de toute fonction, alors elle est globale.
- Si une variable est déclarée à l'intérieur d'une fonction, y compris le programme principal `main`, alors sa portée est limitée à cette fonction.
- Si une variable fait partie des arguments d'une fonction, alors sa portée est limitée à cette fonction.
- Si deux variables de même nom sont déclarées dans deux fonctions différentes, alors elles représentent en réalité deux boîtes distinctes et la portée de chacune est limitée à la fonction correspondante. Dans ce cas, elles peuvent même avoir des types différents.
- Il faut éviter d'utiliser le même nom pour une variable globale et une variable locale. Cependant, si cela se produit, à l'intérieur de la portée de la variable locale, c'est celle-ci qui est visible et non plus la variable globale, qui ne sera à nouveau accessible que quand on sera sorti de la portée de la variable locale.

Exercice 4.6 (avec corrigé)

Le programme ci-après devrait réaliser un générateur de nombres pseudo-aléatoires tel que celui présenté au chapitre 1.

1. Une fonction n'a pas l'effet voulu : laquelle et pourquoi ?
2. Quel problème cela posera-t-il pour l'utilisation de ce programme ?
3. Comment résoudre ce problème ?

```
class Generateur {

  static int valeur;
  static int periode;
  // Cette fonction initialise le générateur
  static void origine(int valeur) {
    int valeurTronquee;
    valeurTronquee = valeur % periode;
    valeur = valeurTronquee;}
  // Cette fonction crée et renvoie un nombre pseudo-aléatoire compris entre 0 et (periode - 1)
  static int hasard() {
    valeur = (15 * valeur + 3) % periode;
    return valeur;}
  // Cette fonction affiche periode valeurs pseudo-aléatoires
  public static void main(String[] args) {
    int i;
    periode = 7;
    origine(8);
    for (i = 1; i <= periode; i = i + 1) {
      System.out.println(hasard());}}}
```

1. *Tel que le programme est écrit, la fonction* `origine` *n'a aucun effet : les deux variables* `valeur` *et* `valeurTronquee` *qu'elle manipule sont locales à cette fonction et les valeurs qui leur sont affectées sont donc effacées de l'état dès la fin de l'exécution de la fonction* `origine`.

2 *La fonction `origine` devrait servir à initialiser le générateur avec la valeur donnée en argument mais, comme on l'a vu, elle ne fait rien en réalité. La variable globale `valeur`, qui est utilisée dans la fonction `hasard`, n'est donc jamais initialisée. Le programme peut alors avoir plusieurs comportements possibles (voir le chapitre 3), notamment en fonction du compilateur utilisé :*
- *La variable `valeur` vaut toujours 0 en début de programme et les nombres engendrés perdent alors leur aspect aléatoire.*
- *La valeur initiale de la variable `valeur` est imprévisible et on ne peut pas maîtriser le générateur de nombres pseudo-aléatoires.*
- *Une erreur peut se produire lors du premier appel à la fonction `hasard`.*

3 *On l'a vu, le problème vient de la coexistence d'une variable globale `valeur` et d'une variable `valeur` locale à la fonction `origine`. La variable globale est ici indispensable vue la façon dont la fonction `hasard` est programmée. Il faut donc réécrire la fonction `origine`, en utilisant un autre nom que le nom `valeur`, par exemple le nom `graine`, comme argument :*

```
static void origine(int graine) {
 int graineTronquee;
 graineTronquee = graine % periode;
 valeur = graineTronquee;}
```

Exercice 4.7 (avec corrigé)

Le programme ci-après est exécuté par un site web pour vérifier que le pseudonyme qu'un utilisateur vient de choisir n'est formé que de lettres minuscules.

1 Déterminer la portée des différentes variables utilisées dans ce programme.

2 Expliquer pourquoi cela ne pose pas de problème d'utiliser le même compteur de boucle `i` dans chacune des deux boucles `for`, alors que la fonction `appartient` est appelée dans le corps de la première boucle.

```
class Pseudo {

// Dans cette fonction, "lettre" est censée ne contenir
// qu'un seul caractère
// On vérifie si ce caractère apparaît dans la chaîne "mot"

static boolean appartient(String lettre, String mot) {
  int i;
  boolean resultat;

  resultat = false;
  for (i = 0; i <= Isn.stringLength(mot) - 1; i = i + 1) {
    if (Isn.stringEqual(Isn.stringNth(mot, i), lettre)) {
      resultat = true;}}
  return resultat;}

public static void main(String[] args) {
  String autorises, pseudo;
  boolean pseudoOk;
  int i;
```

```
  autorises = "abcdefghijklmnopqrstuvwxyz";
  pseudoOk = false;
  // On redemande un pseudo tant qu'il n'est pas correct
  while (!pseudoOk) {
    System.out.println("Entrez votre pseudo :");
    pseudo = Isn.readString();
    pseudoOk = true;
    // On vérifie que chaque caractère du pseudo est autorisé
    for (i = 0; i <= Isn.stringLength(pseudo) - 1; i = i + 1) {
      if (!appartient(Isn.stringNth(pseudo,i),autorises)) {
        pseudoOk = false;}}}}}
```

1. *Les variables* `autorises`, `pseudo` *et* `pseudoOk` *sont locales à la fonction* `main`*. La variable* `resultat` *est locale à la fonction* `appartient`*. Les variables* `mot` *et* `lettre`*, en tant qu'arguments, sont également locales à la fonction* `appartient`*. Il y a deux variables* `i`*, une dans chaque fonction, et chacune est locale à la fonction dans laquelle elle est déclarée.*
2. *Comme il y a en réalité deux variables* `i`*, chacune locale à une fonction, le programme se comporte exactement comme si les deux compteurs de boucle avaient des noms différents. Le compteur de la boucle externe n'est donc pas modifié par la boucle interne. Cela justifie la coutume d'utiliser souvent les variables* `i`*,* `j` *ou* `k` *comme compteurs de boucle quand il n'y a pas de nécessité de leur donner un nom plus explicite.*

Exercice 4.8 (avec corrigé)

Déterminer la portée de chaque variable dans le programme suivant. L'utilisation qui est faite de ces variables est-elle cohérente avec cette portée ? Si non, comment corriger ce programme ?

```
class Portee {
  static int z, y;

  static void v (double x) {
    double u;
    u = x * x;
    z = (int)x;}

  public static void main (String [] args) {
    double t;
    y = 4;
    t = 1.0 / (double)y;
    v(t);
    System.out.println(u);}}
```

Il y a cinq variables dans ce programme : `t`*,* `u`*,* `x`*,* `y` *et* `z`*.* `v` *n'est pas une variable, mais une fonction. Les variables* `z` *et* `y` *sont déclarées avant les fonctions. Elles sont globales et peuvent donc être utilisées partout.*

La variable `x` *est un argument de la fonction* `v` *: sa portée est donc limitée au corps de cette fonction et elle n'est effectivement utilisée que là.*

La variable `u` *est locale à la fonction* `v` *puisqu'elle est déclarée dans le corps de cette fonction, mais elle est utilisée dans le programme principal* `main` *: le programme est donc incorrect.*

Enfin, la variable `t` *est locale au programme principal* `main` *puisqu'elle est déclarée dans le corps de cette fonction. Elle n'est effectivement utilisée que là.*

La dernière ligne du programme affiche la valeur de `u`*. Pour pouvoir le faire tout en respectant la portée des variables, plusieurs solutions existent :*

- *rendre* `u` *globale : c'est peu recommandé, car on essaye de n'utiliser des variables globales que lorsque cela est réellement indispensable ;*
- *déplacer l'affichage de* `u` *à l'intérieur de la fonction* `v` *: cela n'est envisageable que si on est certain qu'à chaque appel de la fonction* `v`*, on voudra afficher la valeur de* `u` *à l'écran ;*
- *modifier la fonction* `v` *pour qu'elle renvoie la valeur de* `u` *: c'est la solution que l'on privilégiera ici puisqu'elle ne modifie pas le comportement intrinsèque de* `v`*. Cependant, cette solution ne serait pas utilisable si* `v` *avait déjà une autre valeur de retour.*

```
class Portee {
  static int z, y;

  static double v (double x) {
    double u;
    u = x * x;
    z = (int)x;
    return u;}

  public static void main (String [] args) {
    double t;
    y = 4;
    t = 1 / (double)y;
    System.out.println(v(t));}}
```

Exercice 4.9

Déterminer la portée de chaque variable dans le programme suivant.

```
class Portee2 {
  static double b;

  static boolean f (int a) {
    int e;
    e = Isn.readInt();
    return (e == a);}

  static double g (double c) {
    double d;
    d = Math.sin (c);
    return d;}

  public static void main (String [] args) {
    b = Isn.readDouble();
    if (f(0)) {
      g(b);}
    else {
      g(-b);}}}
```

Exercice 4.10

Trouver les erreurs de portée dans le programme suivant.

```
class Portee3 {
  static int i;

  static int h (int j) {
    int k;
    j = j + 1;
    System.out.println(i);
    System.out.println(j);
    k = j + i;
    m = m - 1;
    i = 5;
    return k;}

  public static void main (String [] args) {
    int m, n;
    m = 1;
    i = 10;
    System.out.println(m);
    n = h(m);
    System.out.println(m);
    System.out.println(k);
    System.out.println(i + j);}}
```

SAVOIR-FAIRE Choisir une portée adaptée aux différentes variables d'un programme comportant des fonctions

Pour chacune de ces variables :

- si elle est utilisée dans plusieurs fonctions, alors elle doit être globale,
- si elle n'est utilisée que dans une fonction, pour des calculs intermédiaires par exemple, on préfère qu'elle soit locale à cette fonction.

Exercice 4.11 (avec corrigé)

Où faut-il placer les déclarations des différentes variables dans le programme suivant ?

```
class Repertoire {

  static void initialisation () {
    nb = 10;
    nom = new String [10];
    tel = new String [10];
    nom[0] = "Alice";
    tel[0] = "0606060606";
    nom[1] = "Bob";
    tel[1] = "0606060607";
```

```
    nom[2] = "Charles";
    tel[2] = "0606060608";
    nom[3] = "Djamel";
    tel[3] = "0606060609";
    nom[4] = "Étienne";
    tel[4] = "0606060610";
    nom[5] = "Frédérique";
    tel[5] = "0606060611";
    nom[6] = "Guillaume";
    tel[6] = "0606060612";
    nom[7] = "Hector";
    tel[7] = "0606060613";
    nom[8] = "Isabelle";
    tel[8] = "0606060614";
    nom[9] = "Jérôme";
    tel[9] = "0606060615";}

  static String recherche (String s) {
    i = 0;
    while (i < nb && !Isn.stringEqual(s,nom[i])) {
      i = i + 1;}
    if (i < nb) {
      r = tel[i];}
    else {
      r = "Inconnu";}
    return r;}

  public static void main (String [] args) {
    initialisation ();
    n = Isn.readString();
    System.out.println(recherche(n));}}
```

Les variables `i` *et* `r` *ne sont utilisées que dans la fonction* `recherche` *; elles peuvent donc être locales à cette fonction. La variable* `s` *est un argument de cette fonction ; elle est donc déclarée comme argument. La variable* `n` *n'est utilisée que dans le programme principal ; elle peut donc être locale à cette fonction. Les variables* `nb`, `nom` *et* `tel` *qui contiennent l'information sur le répertoire doivent en revanche être globales. On notera que la fonction* `initialisation` *n'a pas besoin d'arguments puisqu'elle n'utilise que des variables globales.*

```
class Repertoire {

  static String [] nom, tel;
  static int nb;

  static void initialisation () {
    nb = 10;
    nom = new String [10];
    tel = new String [10];
    nom[0] = "Alice";
    tel[0] = "0606060606";
    nom[1] = "Bob";
    tel[1] = "0606060607";
```

```
    nom[2] = "Charles";
    tel[2] = "0606060608";
    nom[3] = "Djamel";
    tel[3] = "0606060609";
    nom[4] = "Étienne";
    tel[4] = "0606060610";
    nom[5] = "Frédérique";
    tel[5] = "0606060611";
    nom[6] = "Guillaume";
    tel[6] = "0606060612";
    nom[7] = "Hector";
    tel[7] = "0606060613";
    nom[8] = "Isabelle";
    tel[8] = "0606060614";
    nom[9] = "Jérôme";
    tel[9] = "0606060615";}

  static String recherche (String s) {
    int i;
    String r;
    i = 0;
    while (i < nb && !Isn.stringEqual(s,nom[i])) {
      i = i + 1;}
    if (i < nb) {
      r = tel[i];}
    else {
      r = "Inconnu";}
    return r;}

  public static void main (String [] args) {
    String n;
    initialisation ();
    n = Isn.readString();
    System.out.println(recherche(n));}}
```

Le passage par valeur

Dans le programme suivant :

```
static int a,b;

public static void main (String [] args) {
  int c;
  a = 4;
  b = 7;
  c = a;
  a = b;
```

```
  b = c;
  System.out.print(a);
  System.out.print("   ");
  System.out.println(b);}
```

l'exécution de l'instruction :

```
c = a;
a = b;
b = c;
```

a pour effet d'échanger le contenu des boîtes a et b. Le contenu initial de la boîte a est 4 et celui de la boîte b 7 ; après l'exécution de cette instruction, le contenu de la boîte a est 7 et celui de la boîte b 4. Ainsi le programme affiche :

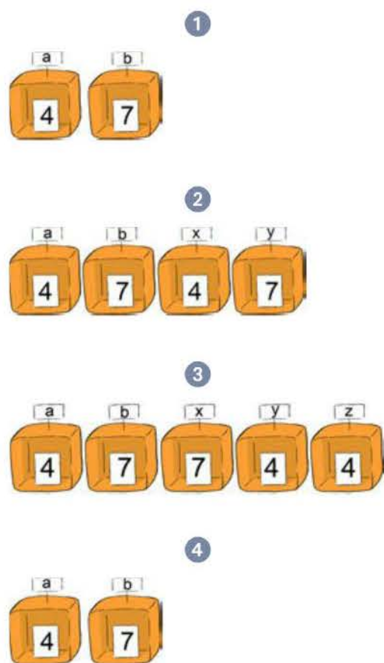
```
7   4
```

Cette opération d'échange du contenu de deux boîtes étant souvent utilisée, on peut vouloir l'isoler dans une fonction.

```
static int a,b;

static void echange (int x, int y) {
  int z;
  z = x;
  x = y;
  y = z;}

public static void main (String [] args) {
  a = 4;
  b = 7;
  echange(a,b);
  System.out.print(a);
  System.out.print("   ");
  System.out.println(b);}
```



Toutefois, contrairement au précédent, ce programme affiche :

```
4   7
```

et non :

```
7   4
```

En effet, l'appel de la fonction echange(a,b); dans l'état ❶ crée l'état ❷, crée une boîte z, échange le contenu des boîtes x et y en utilisant la boîte z (❸) et supprime la boîte z, puis les boîtes x et y (❹).

Le contenu des boîtes a et b n'a donc pas changé.

Il faut donc se souvenir que, dans un appel `echange(e,e');`, on ne passe jamais les expressions `e` et `e'` à la fonction `echange` ; on ne passe que la valeur de ces expressions. Ces valeurs sont mises dans de nouvelles boîtes de noms `x` et `y`, qui n'existent que le temps de l'exécution du corps de la fonction. Quand les expressions `e` et `e'` sont des variables `a` et `b`, cela revient à recopier le contenu des boîtes de noms `a` et `b` dans les boîtes de noms `x` et `y`.

SAVOIR-FAIRE

Choisir entre un passage par valeur et une variable globale

Si une fonction doit utiliser une variable `a` du programme principal, deux cas sont à distinguer :

- Si la fonction n'utilise que la valeur contenue dans `a` sans jamais la modifier, alors cette valeur peut être passée en argument à la fonction et la variable `a` peut être locale au programme principal.
- Si la fonction doit modifier la valeur contenue dans `a`, alors la variable `a` doit être globale.

Exercice 4.12 (avec corrigé)

Dans le programme suivant, quelles sont les expressions passées par valeur ?
Qu'affiche ce programme lorsqu'on l'exécute ?

```
class ParValeur {
  static int i;

  static int h (int j) {
    int k;
    j = j + 1;
    System.out.println(i);
    System.out.println(j);
    k = j + i;
    i = 5;
    return k;}

  public static void main (String [] args) {
    int m, n;
    m = 1;
    i = 10;
    System.out.println(m);
    n = h(m);
    System.out.println(m);
    System.out.println(n);
    System.out.println(i);}}
```

Lors de l'appel `h(m)`, la valeur 1, contenue à ce moment dans la boîte `m`, est passée à la fonction `h` via l'argument `j`. La variable `m` n'est pas modifiée, même par l'instruction `j = j + 1;` et elle conserve sa valeur 1 jusqu'à la fin du programme principal.

Ce programme affiche donc :

1	*Valeur initiale de* `m`
10	*La variable globale* `i` *a été initialisée à 10 dans le programme principal.*
2	`j` *prend la valeur 1 de* `m` *lors de l'appel à* `h`*, puis on lui ajoute 1.*
1	`m` *n'est pas modifiée par* `h`*.*
12	`n` *contient la valeur de retour de* `h`*, qui a été donnée par l'expression* `j + i`*, dans laquelle* `j` *valait 2 et* `i` *valait 10.*
5	*On a affecté la valeur 5 à la variable globale* `i` *dans la fonction* `h`*.*

Exercice 4.13

Pour programmer les fonctions ci-dessous, faut-il plutôt utiliser un passage par valeur ou des variables globales ?

1. Une fonction qui trace à l'écran un segment connaissant les coordonnées de ses extrémités.
2. Une fonction qui remplace un nombre par sa valeur absolue.
3. Une fonction qui calcule le logarithme entier d'un nombre.
4. Une fonction qui met en majuscules toutes les lettres d'une chaîne de caractères.
5. Une fonction qui affiche à l'écran en notation scientifique un nombre donné de type `double`.

Le passage par valeur et les tableaux

Contrairement à celui de la section précédente, le programme suivant, qui utilise des tableaux, affiche bien `7 4`.

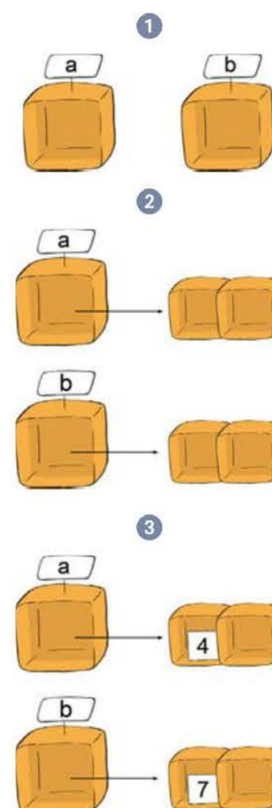
```
static int [] a,b;

static void echange (int [] x, int [] y) {
  int z;
  z = x[0];
  x[0] = y[0];
  y[0] = z;}

public static void main (String [] args) {
  a = new int [2];
  b = new int [2];
  a[0] = 4;
  b[0] = 7;
```

```
echange(a,b);
System.out.print(a[0]);
System.out.print("   ");
System.out.println(b[0]);}
```

En effet, comme on l'a vu au chapitre 3, la déclaration des variables globales a et b, qu'on écrit `static int [] a,b;` ajoute deux boîtes de noms a et b à l'état ❶. Puis, l'allocation des deux tableaux :



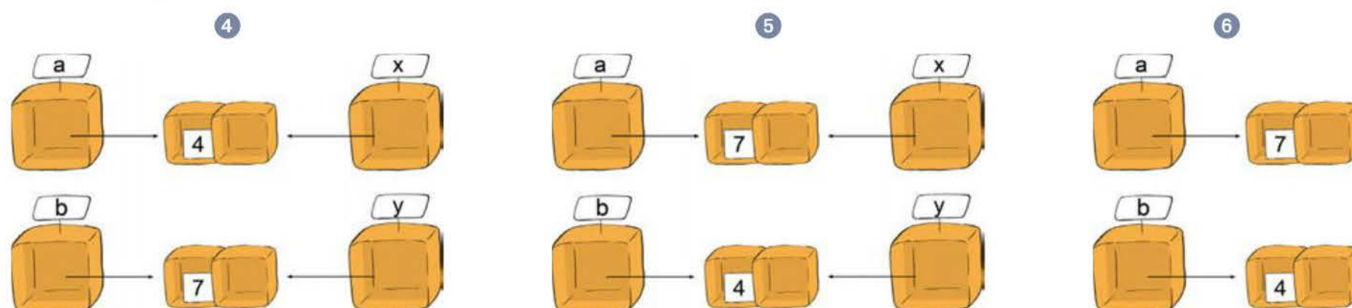

```
a = new int [2];
b = new int [2];
```

crée deux nouvelles boîtes (❷). L'affectation qui suit :

```
a[0] = 4;
b[0] = 7;
```

remplit les cases 0 de ces deux tableaux avec les valeurs 4 et 7 (❸). Ensuite, l'appel de la fonction `echange(a,b);` crée deux boîtes x et y et les remplit avec la valeur des expressions a et b. Or, comme on l'a vu au chapitre 3, la valeur de l'expression a est une référence vers la boîte à deux cases située à gauche sur la figure et celle de l'expression b est une référence vers la boîte à deux cases située à droite sur la figure. Ainsi, l'état construit est ❹. L'exécution du corps de la fonction échange le contenu des cases x[0] et y[0], ce qui produit l'état ❺. Et quand, à la fin de l'exécution du corps de la fonction, les boîtes de noms x et y sont supprimées, l'état ❻ est produit. La valeur de l'expression a[0] est donc 7 et celle de l'expression b[0] est 4.

Autrement dit, le contenu des boîtes de noms a et b est bien recopié dans les boîtes de noms x et y au moment de l'appel. Néanmoins, les tableaux eux-mêmes ne sont pas recopiés, car les contenus des boîtes de noms a et b ne sont pas des tableaux mais des références de tableaux.





Exercice 4.14

Reprendre l'explication précédente dans le cas où les deux variables a et b sont locales au programme principal.

ALLER PLUS LOIN Les licences logicielles : logiciel libre *versus* logiciel propriétaire

Quand on écrit un programme qui est utilisé par d'autres personnes que soi, on doit préciser ses conditions d'utilisation par un contrat qui s'appelle une *licence logicielle*.

Les programmes ont certains points communs avec d'autres biens non rivaux (voir le chapitre 11), comme les inventions et les « œuvres de l'esprit ». Cela explique que le droit des licences ait certains points communs avec le droit des brevets et le droit d'auteur. Le droit des brevets et le droit d'auteur donnent aux inventeurs et aux auteurs le monopole de l'exploitation de leur création et les autorisent à vendre ce droit d'exploitation. Ils favorisent donc la création, en permettant aux créateurs de gagner de l'argent. Cependant, ils la freinent également, car ils interdisent à d'autres d'utiliser ces créations et de poursuivre le travail de leurs auteurs. C'est pour cela que ce monopole d'exploitation est toujours limité dans le temps et que, dans le droit des brevets, l'inventeur est souvent tenu de rendre à terme son invention publique.

Il y a principalement deux formes de licences logicielles : les *licences propriétaires* (ou privatrices) et les *licences libres*. Avec une licence propriétaire, l'auteur donne simplement un droit d'utilisation de son programme. La plupart du temps, il diffuse le code compilé de son programme, mais en garde le code source (voir le chapitre 15) secret. Avec une licence libre, en revanche, il donne le droit à ses utilisateurs non seulement d'utiliser son programme, mais aussi d'en étudier le fonctionnement et de le modifier. Il diffuse donc à la fois le code compilé et le code source.

Parmi les licences libres, il y a encore deux types de licences. Les licences *contaminantes*, comme la *General Public Licence* (GPL), imposent, dans le cas de la diffusion du programme modifié, de le diffuser avec la même licence, afin de faire bénéficier les autres des mêmes libertés que celles dont on a soi-même bénéficié. D'autres licences, comme la licence *Berkeley Software Distribution* (BSD), donnent la liberté de modifier le programme et de diffuser la version modifiée sous une licence propriétaire.

Les licences libres ont permis un nouveau mode de développement des logiciels, par une démarche qui a un certain nombre de points communs avec la recherche scientifique : coopération internationale, publication, validation par les pairs, liberté de critiquer et de modifier, etc., alors que, dans les temps anciens, Pythagore, par exemple, interdisait à ses disciples de divulguer leurs théorèmes et leurs démonstrations. Au-delà des programmes, cette nouvelle manière de produire et de diffuser des objets industriels préfigure peut-être une évolution plus globale de l'industrie, dans un monde dans lequel de plus en plus de biens industriels sont complexes et immatériels.

Ai-je bien compris ?

- À quoi sert une fonction ?
- De quels éléments la définition d'une fonction est-elle composée ?
- Comment utilise-t-on une fonction dans un programme ?

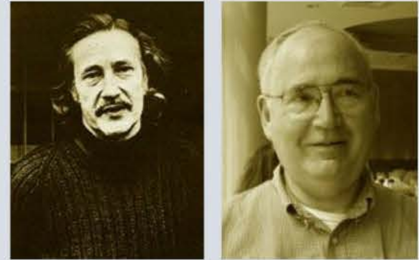
La récursivité

5

Chapitre avancé

Une définition récursive est une définition récursive.

Dans ce chapitre, nous voyons qu'une fonction peut s'appeler elle-même. Cette construction, alternative à celle de boucle, permet d'écrire des programmes courts et élégants.



Christopher Strachey (1916-1975) et **Dana Scott** (1932-) ont donné une sémantique aux définitions récursives : la définition $f = G(f)$ n'est pas circulaire, mais une définition de f comme point fixe de G. Christopher Strachey est aussi l'auteur d'un des premiers programmes jouant aux dames et de l'un des premiers programmes d'informatique musicale. Avec Michael Rabin, Dana Scott a étudié les systèmes à états et transitions (voir le chapitre 22) *non déterministes*, c'est-à-dire dans lesquels plusieurs transitions sont possibles à partir d'un même état.

Des fonctions qui appellent des fonctions

Au chapitre 4, nous avons vu que les fonctions permettaient d'isoler une instruction du programme principal. Par exemple, au lieu d'écrire le programme :

```
static public void main (String [] args) {
  System.out.print("Le vol en direction de ");
  System.out.print("Tokyo");
  System.out.print(" décollera à ");
  System.out.print("9h00");
  System.out.println();
  System.out.println("--------------------------------------------");
  System.out.println();

  System.out.print("Le vol en direction de ");
  System.out.print("Sydney");
  System.out.print(" décollera à ");
  System.out.print("9h30");
  System.out.println();
  System.out.println("--------------------------------------------");
  System.out.println();

  System.out.print("Le vol en direction de ");
  System.out.print("Toulouse");
  System.out.print(" décollera à ");
  System.out.print("9h45");
  System.out.println();
  System.out.println("--------------------------------------------");
  System.out.println();}
```

il est possible d'écrire le programme :

```
static void annoncerUnVol (String destination, String horaire) {
  System.out.print("Le vol en direction de ");
  System.out.print(destination);
  System.out.print(" décollera à ");
  System.out.print(horaire);
  System.out.println();
  System.out.println("--------------------------------------------");
  System.out.println();}

static public void main (String [] args) {
  annoncerUnVol("Tokyo","9h00");
  annoncerUnVol("Sydney","9h30");
  annoncerUnVol("Toulouse","9h45");}
```

Il est aussi possible d'isoler l'instruction :

```
System.out.println();
System.out.println("------------------------------------------");
System.out.println();
```

de la fonction `annoncerUnVol` et d'écrire ce programme ainsi :

```
static void tirerUnTrait () {
  System.out.println();
  System.out.println("------------------------------------------");
  System.out.println();}

static void annoncerUnVol (String destination, String horaire) {
  System.out.print("Le vol en direction de ");
  System.out.print(destination);
  System.out.print(" décollera à ");
  System.out.print(horaire);
  tirerUnTrait ();}

static public void main (String [] args) {
  annoncerUnVol("Tokyo","9h00");
  annoncerUnVol("Sydney","9h30");
  annoncerUnVol("Toulouse","9h45");}
```

où la fonction `tirerUnTrait` est appelée, non depuis le programme principal, mais depuis le corps de la fonction `annoncerUnVol`.

Des fonctions qui s'appellent elles-mêmes

Il est même possible d'aller plus loin et d'appeler une fonction `f`, non depuis le corps d'une fonction `g`, mais depuis le corps de la fonction `f` elle-même. C'est, par exemple, le cas de la fonction suivante :

```
static int puissance (int exposant) {
  if (exposant == 0) {
    return 1;}
  else {
    return (2 * puissance(exposant - 1));}}
```

Fonction récursive

Une fonction récursive est une fonction qui s'appelle elle-même.

Le cas le plus simple est celui de l'appel `puissance(0)` qui retourne la valeur 1 sans avoir besoin de refaire un appel à la fonction `puissance`.

On peut alors comprendre la valeur retournée par l'appel `puissance(1)` : quand on exécute le corps de la fonction dans un état dans lequel la boîte de nom `exposant` contient la valeur 1, on évalue l'expression `2 * puissance(exposant - 1)`. Comme la valeur de l'expression `exposant` est 1, celle de l'expression `exposant - 1` est 0 et celle de l'expression `puissance(exposant - 1)` est, comme on l'a vu, 1. Celle de l'expression `2 * puissance(exposant - 1)` est donc 2. L'appel `puissance(1)` retourne donc la valeur 2.

De même, l'appel `puissance(2)` retourne la valeur 4, l'appel `puissance(3)` retourne la valeur 8, et ainsi de suite.

> ATTENTION **Prévoir un cas de base**
>
> Dans la définition d'une fonction récursive, il faut toujours prévoir au moins un *cas de base*, comme le cas `exposant == 0` dans la définition ci-avant, dans lequel la fonction ne s'appelle pas elle-même ; sinon elle s'appellera elle-même indéfiniment.

Plus généralement, la valeur retournée par l'appel de la fonction `puissance` avec l'argument $k + 1$ est le double de celle retournée par l'appel de la fonction `puissance` avec l'argument k. La valeur retournée par l'appel de la fonction `puissance` avec l'argument k est donc 2^k.

Cette fonction calcule donc la même chose que la fonction :

```
static int puissance (int exposant) {
  int i,resultat;
  resultat = 1;
  for (i = 1; i <= exposant; i = i + 1) {
    resultat = 2 * resultat;}
  return resultat;}
```

Toutefois, elle utilise cette possibilité pour une fonction de s'appeler elle-même, au lieu d'utiliser une boucle.

On note que dans la définition récursive de la fonction `puissance`, la valeur de l'argument `exposant` diminue à chaque appel de la fonction et le test effectué au début du corps de la fonction assure que la fonction `puissance` ne s'appelle pas elle-même au cours de son exécution dans un état dans lequel la boîte de nom `exposant` contient la valeur 0. Cette observation fournit la garantie que tout appel à la fonction `puissance` avec un argument qui est un nombre entier positif `exposant` se termine après `exposant` appels. Les fonctions récursives et celles utilisant des boucles partagent ainsi la même préoccupation de terminaison.

Savoir-faire Définir une fonction récursive

1 Écrire l'en-tête de la fonction (voir le savoir-faire, page 60, « Écrire l'en-tête d'une fonction »).

2 Vérifier tout d'abord que la fonction est adaptée à une définition récursive, autrement dit que l'on sait calculer une valeur à partir d'un appel plus simple à la même fonction.

3 Prévoir le ou les cas de base, qui ne nécessitent pas d'appel récursif de la fonction.

4 Dans tous les appels récursifs, s'assurer que les arguments sont plus simples que ceux avec lesquels la fonction a été appelée : nombres plus petits, chaînes de caractères plus courtes, etc.

5 Reconstituer correctement la valeur de retour de la fonction à partir du résultat du ou des appel(s) récursif(s).

Exercice 5.1 (avec corrigé)

Écrire une fonction récursive qui calcule le quotient de la division euclidienne d'un nombre entier naturel par un autre, bien entendu sans utiliser l'opération / du langage.

1 *Le dividende et le diviseur sont les deux arguments et la valeur de retour est le quotient. On a donc l'en-tête :*

```
static int quotient (int dividende, int diviseur)
```

2 *Une définition récursive est possible : on diminue le dividende à chaque appel récursif jusqu'à avoir compté combien de fois il contient le diviseur.*

3 *Le cas de base est celui où le dividende est inférieur au diviseur : le quotient est alors nul. La fonction commence donc par le test :*

```
if (dividende < diviseur) {
  return 0;}
```

4 *Dans les autres cas, le dividende est supérieur au diviseur, on retranche le diviseur au dividende et l'argument ainsi modifié reste positif. Le diviseur n'est pas modifié. La fonction se termine toujours car le dividende, qui est un entier positif, ne peut pas diminuer indéfiniment.*

5 *À chaque appel récursif, on compte une fois le diviseur dans le dividende, le quotient doit donc augmenter de 1. La fonction s'écrit donc :*

```
static int quotient (int dividende, int diviseur) {
  if (dividende < diviseur) {
    return 0;}
  else {
    return 1 + quotient (dividende - diviseur, diviseur);}}
```

Exercice 5.2

Modifier le programme ci-avant pour qu'il calcule, non le quotient, mais le reste d'une division euclidienne.

Exercice 5.3

Écrire une fonction récursive qui calcule le logarithme entier d'un nombre (voir le chapitre 2).

Exercice 5.4

Écrire une fonction récursive qui calcule la factorielle d'un nombre $n! = 1 \times 2 \times \ldots \times (n-1) \times n$.



Exercice 5.5

Écrire une fonction récursive qui calcule le plus grand diviseur commun (PGCD) de deux nombres entiers, en utilisant l'algorithme d'Euclide. Dans ce programme, en quoi les arguments de l'appel récursif sont-ils plus simples que ceux avec lesquels la fonction est appelée ?



Exercice 5.6

Programmer récursivement le calcul du terme de rang n de la suite de Fibonacci définie par :

1. $u_0 = u_1 = 1$
2. $u_{n+2} = u_n + u_{n+1}$

Qu'y a-t-il de particulier dans cette fonction récursive ? Que se passe-t-il si on exécute ce programme pour de « grandes » valeurs de n, à partir de 35 ou 45 selon la machine utilisée ? Par ailleurs, indépendamment de la machine utilisée, à partir de $n = 46$, les valeurs données par ce programme deviennent incorrectes. Expliquer ces deux phénomènes.

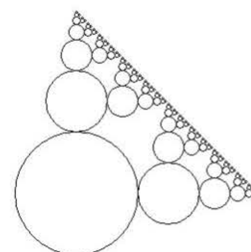
ALLER PLUS LOIN L'efficacité des fonctions récursives

Comme on a pu le voir sur l'exemple de la suite de Fibonacci, si les définitions récursives rendent parfois très pratiques l'écriture d'un programme, elles peuvent aussi avoir des conséquences très néfastes sur leur efficacité. Ainsi, dans notre exemple, pour calculer u_{10}, il faut calculer u_9 et u_8, mais comme u_9 est calculé récursivement, pour l'obtenir il faut calculer u_8 et u_7. Le calcul de u_8 est donc fait deux fois. On peut montrer de même que u_7 est calculé trois fois, u_6 cinq fois, etc.
Cela n'est pas un problème inhérent aux fonctions récursives, c'est juste un piège dans lequel on peut facilement tomber quand on en écrit une. Il faut dans un tel cas se demander si certains calculs ne sont pas faits plusieurs fois, et si c'est le cas comment l'éviter. Les solutions habituelles sont de mémoriser, d'une façon ou d'une autre, les calculs déjà effectués et qui serviront à nouveau. Dans le cas de la suite de Fibonacci, il suffit de calculer les termes de la suite dans l'ordre en se souvenant des deux dernières valeurs calculées. Avec les fonctions récursives, on appelle *accumulateurs* les arguments qui propagent ces deux valeurs d'un appel de la fonction au suivant.

```
static int fib (int n, int u1, int u0) {
  if (n > 1) {
    return fib(n - 1,u1 + u0,u1);}
  else {
    if (n == 1) {
      return u1;}
    else {
      return u0;}}}
```

Cette fonction permet en réalité de calculer le n-ème terme de toute suite vérifiant la même relation de récurrence que la suite de Fibonacci, en spécifiant les valeurs de u_0 et u_1. Pour le n-ème terme de la suite de Fibonacci elle-même, il suffit d'appeler `fib(n,1,1)`.

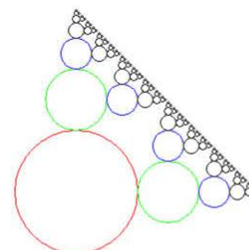
Des images récursives



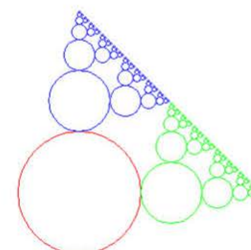
La récursivité est un outil utile en géométrie algorithmique et en synthèse d'images. Observons, par exemple, le dessin ci-contre.

Une manière de le décrire est de dire qu'il est formé :

- d'un cercle, en rouge sur la figure ci-contre,
- en haut et à droite de ce cercle, de deux cercles tangents de rayon moitié moindre, en vert sur la figure,
- en haut et à droite de chacun de ces cercles, de deux cercles tangents de rayon moitié moindre, en bleu sur la figure,
- en haut et à droite de chacun de ces cercles, de deux cercles tangents de rayon moitié moindre,
- etc.



Toutefois, une autre manière de le décrire est de dire qu'il est formé d'un cercle, en rouge sur la figure ci-contre et de deux copies moitié plus petites du dessin lui-même, en vert et en bleu sur la figure.



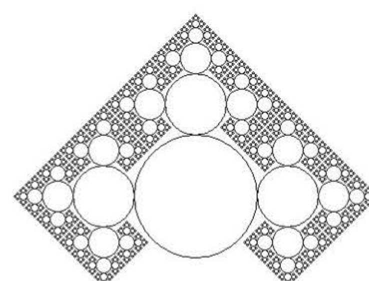
Cela nous montre comment écrire une fonction récursive qui dessine cette figure : pour dessiner une figure dont le plus grand cercle a le centre `(x ; y)` et le rayon `rayon`, on trace d'abord ce cercle, puis on appelle récursivement deux fois la fonction `dessiner` en centrant le grand cercle en (`x + 3 * rayon / 2 ; y`) et en (`x ; y - 3 * rayon / 2`) et en lui donnant le rayon `rayon / 2`.

```
static void dessiner (int x, int y, int rayon) {
  Isn.drawCircle(x,y,rayon,0,0,0);
  if (rayon > 1) {
    dessiner(x + 3 * rayon / 2,y,rayon / 2);
    dessiner(x,y - 3 * rayon / 2,rayon / 2);}}
```

Il ne reste plus qu'à appeler dans le programme principal la fonction `dessiner(200,200,64);`.

Exercice 5.7

Quel est l'argument qui assure que cette fonction ne s'appelle elle-même qu'un nombre fini de fois ? Avec l'appel initial proposé, combien d'appels récursifs auront lieu ?



Un dessin plus joli est obtenu en entourant chaque cercle, non pas de deux, mais de trois dessins plus petits dans des directions qui dépendent de sa position : s'il est lui même à droite d'un cercle plus grand, par exemple, alors il est formé d'un cercle et de trois dessins situés en haut, en bas et à droite, mais pas à gauche, de ce cercle. Pour cela, on fait correspondre un nombre à chaque direction et on ajoute un argument à la fonction.

```
// Définition des directions
static int gauche = 0;
static int droite = 1;
static int haut   = 2;
static int bas    = 3;
static int aucun  = 4;

// Dessin
static void dessiner (int x, int y, int rayon, int v) {
  Isn.drawCircle(x,y,rayon,0,0,0);
  if (rayon > 1) {
    if (v != droite) {
      dessiner(x + 3 * rayon / 2,y,rayon / 2,gauche);}
    if (v != gauche) {
      dessiner(x - 3 * rayon / 2,y,rayon / 2,droite);}
    if (v != haut) {
      dessiner(x,y - 3 * rayon / 2,rayon / 2,bas);}
    if (v != bas) {
      dessiner(x,y + 3 * rayon / 2,rayon / 2,haut);}}}
```

Exercice 5.8

Que représente le nouvel argument v dans cette fonction ? Quel nom serait plus explicite pour cet argument ?



Exercice 5.9

Comment utiliser, sans la modifier, cette dernière fonction pour obtenir le dessin complet où le cercle central est entouré de quatre motifs identiques en haut, en bas, à droite et à gauche ?

Ai-je bien compris ?

- Qu'est-ce qu'une fonction récursive ?
- Quelle autre construction la récursivité permet-elle souvent de remplacer ?
- Que faut-il toujours prévoir dans la définition d'une fonction récursive, si on veut que cette fonction se termine ?

La notion de langage formel

6

Chapitre avancé



L'*idéographie* de **Gottlob Frege** (1848 - 1925) est le premier langage formel proposé pour exprimer l'ensemble des mathématiques. Ce langage a été aujourd'hui abandonné, mais il est à l'origine de la théorie des ensembles, de la logique des prédicats et, en grande partie, des langages formels utilisés en informatique, en particulier des langages de programmation. L'origine commune des mots « langage », « logique » et « logiciel » nous rappelle les liens entre la logique – l'étude du langage mathématique – et l'informatique.

Les langages informatiques sont presque comme les langues naturelles. Mais pas tout à fait.

Dans ce chapitre, hors programme à l'exception de la section « Redéfinir la sémantique », nous introduisons la notion de langage formel, au-delà des langages de programmation, et nous comparons ces langages aux langues naturelles. Nous prenons l'exemple du langage HTML, présenté au chapitre 8. Ceci nous permet de présenter les notions de grammaire et de sémantique.

La grammaire définit quelles sont les chaînes de caractères, par exemple les programmes, qui sont correctement formés.

La sémantique définit ce qui se passe quand on utilise un texte, par exemple quand on exécute un programme.

Aux chapitres 1 à 5, 15, 8 et 11, nous avons vu plusieurs langages : des langages de programmation tout d'abord, mais aussi, plus brièvement, des langages de description de pages web et des langages d'interrogation de bases de données. Ces langages informatiques ont un certain nombre de points communs avec les langues naturelles comme le français ou le japonais : ils servent à exprimer et à communiquer des idées. Cependant, ils ont aussi un certain nombre de différences.

Les langages informatiques et les langues naturelles

Nous avons vu qu'un langage de programmation, comme Java, permet d'exprimer tous les algorithmes. Toutefois, il ne permet d'exprimer que des algorithmes. On ne peut pas l'utiliser pour écrire un roman, un contrat ou une carte postale. C'est donc un langage *spécialisé*. Une langue naturelle, en revanche, peut être utilisée non seulement pour exprimer des algorithmes, même si c'est souvent de manière imprécise, mais aussi pour écrire des cartes postales, des contrats et des romans : les langues naturelles sont *universelles*.

Le vocabulaire d'une langue naturelle contient plusieurs milliers de mots. Celui d'un langage informatique, par exemple un langage de programmation, n'en contient que quelques dizaines : par exemple `if`, `while`, `for` et `int`. En revanche, le vocabulaire d'un langage informatique est souvent extensible : en définissant une fonction `tirerUnTrait`, ou en déclarant une variable `x`, on ajoute un nouveau mot au langage, souvent avec une portée limitée, ce que l'on ne peut pas faire dans une langue naturelle.

ALLER PLUS LOIN

La grammaire des langues naturelles

La grammaire des langues naturelles est beaucoup plus complexe que celle des langages formels, en particulier parce qu'elle tient compte de nombreuses exceptions : homonymie, synonymie, etc. Ainsi, en français, la phrase « les poules du couvent couvent » est correcte.

La grammaire des langages informatiques est plus simple, mais plus précise, que celle des langues naturelles : un point-virgule oublié dans un programme et celui-ci n'est plus correct, alors qu'il est difficile de trouver une phrase en français qui serait incompréhensible à cause d'un signe de ponctuation oublié. De plus, la frontière entre les phrases correctes et les phrases incorrectes dans une langue naturelle n'est pas si bien définie. Ainsi, la phrase « Nous, on y est pas allé, mon frère et moi, au cinéma. » manifeste une moins grande maîtrise de la langue que la phrase « Ni mon frère ni moi ne sommes allés au cinéma. », mais elle reste compréhensible. En revanche, il n'y a aucune discussion possible sur le fait qu'un pro-

gramme soit grammaticalement correct ou non, car la correction grammaticale, dans un langage de programmation, est définie par un algorithme.

Enfin, ces langages informatiques appartiennent exclusivement au domaine de l'écrit : une pièce de théâtre est écrite pour être dite à haute voix, pas un programme.

Les ancêtres des langages formels

Pour les distinguer des langues naturelles, on appelle *langages formels* ces langages artificiels à la grammaire simple, mais précise. L'apparition, avec l'informatique, de ces langages est une étape importante de l'histoire du langage. Certains la comparent même à l'invention de l'écriture ou de l'alphabet. Cependant, une fois que l'on a pris conscience de la spécificité de ces langages, on peut se demander si certains de leurs traits n'étaient pas déjà présents dans des langages spécialisés plus anciens, en particulier dans ceux utilisés en sciences, en droit et en musique.

Tout d'abord, sans cesser d'être une langue naturelle, la langue scientifique présente quelques aspects qui rappellent les langages informatiques. Par exemple, elle donne la possibilité d'introduire de nouveaux mots par des définitions. S'il n'est pas possible dans la langue courante de dire « On appellera *épagneul breton* un steak saignant. », puis « Je voudrais manger un épagneul breton. », on peut, dans la langue scientifique, donner une définition : « On appellera *travail* d'une force, son produit scalaire avec le déplacement de son point d'application. », puis utiliser le mot défini : « Le travail d'une force orthogonale au déplacement de son point d'application est nul. » Une fois défini, le mot « travail » a une portée illimitée, mais ce n'est pas le cas du mot x si on l'introduit dans une démonstration mathématique par la définition $x = y + 4$. Une fois cette démonstration achevée, il n'est plus possible d'utiliser le mot x pour désigner la valeur $y + 4$. De même, dans le langage juridique, on débute le texte d'un contrat en donnant l'identité des contractants et en indiquant un mot par lequel ils seront désignés dans la suite du contrat. Par exemple, « Contrat de location entre les sous-signés, Monsieur Dupond, demeurant 1, rue Durand, à Grenoble, ci-après désigné le loueur et Monsieur Durand, demeurant 1, rue Dupond, à Grenoble, ci-après désigné le locataire … ». Bien entendu, cette manière de désigner Mon-

sieur Dupond et Monsieur Durand comme « le loueur » et « le locataire » a une portée limitée au texte de ce contrat.

Cependant, il y a aussi des situations où l'on s'éloigne davantage des langues naturelles et crée de véritables langages formels, au vocabulaire limité et à la grammaire simple mais précise. Par exemple, la nomenclature des composés chimiques, la notation musicale et le langage algébrique. Ainsi, dans la nomenclature des composés chimiques, il y a un chlorure d'aminométhylpyrimidinylhydroxyéthylméthythiazolium, mais pas de chlorure d'épagneul breton : seul un nombre restreint de mots peut être employé dans ce langage. Comme la nomenclature des composés chimiques, la notation musicale utilise un vocabulaire restreint et une grammaire simple, mais précise : elle autorise à mettre six croches dans une mesure binaire à trois temps, mais pas quatre noires. De même, le langage algébrique permet d'appliquer la relation < à deux expressions, mais pas à la relation < elle-même : on peut écrire $x^2 < 4$, mais pas $x^2 < <$.

Aller plus loin Mélanger langages formels et langues naturelles

Souvent, dans un texte scientifique, on mélange des passages écrits dans des langages formels, comme le langage algébrique, et des passages écrits en langue naturelle : « Supposons qu'il existe deux nombres entiers non nuls et premiers entre eux tels que $x^2 = 2\,y^2$. Le nombre x^2 est pair et, d'après le lemme 1, le nombre x également : il existe donc un nombre x' tel que $x = 2\,x'$. On en déduit $(2\,x')^2 = 2\,y^2$, c'est-à-dire $4\,x'^2 = 2\,y^2$, d'où on tire $y^2 = 2\,x'^2$. Le nombre y^2 est donc pair et, en utilisant le lemme 1 à nouveau, le nombre y également. Les nombres x et y sont tous les deux pairs, ce qui est en contradiction avec l'hypothèse ».

Ces langages formels sont relativement récents : la notation musicale date du XIIIe siècle, la notation algébrique du XVIe siècle et la nomenclature des composés chimiques du XIXe siècle. Avant l'invention de la notation algébrique, on écrivait les équations en langue naturelle, par exemple l'équation $x^3 + 3\,x^2 = 20$ s'écrivait : un cube et trois carrés font vingt. Néanmoins, ces langages sont tous les trois antérieurs à l'informatique.

La rançon de l'universalité des langues naturelles semble donc être leur incapacité à exprimer précisément les choses, dès que l'on s'intéresse à un domaine précis comme les sciences, le droit ou la musique. Raison pour laquelle les scientifiques, les juristes ou les musiciens se sont éloignés des langues naturelles, parfois par de petits écarts, parfois en créant de véritables langages formels. L'apparition et la généralisation, avec l'informatique, des langages formels étaient donc préparées par le développement de ces langages spécialisés, plus ou moins formels.

Les langages formels et les machines

Les langages de programmation, et plus généralement les langages informatiques, sont soumis à une double contrainte. Ils doivent être utilisables par les êtres humains qui écrivent les programmes, mais aussi par les machines qui les exécutent. Cette seconde contrainte explique en

partie que ce soient des langages formels. Au début de l'informatique, cette seconde contrainte était prépondérante et seuls existaient les langages machine, comme celui décrit au chapitre 15, ce qui rendait l'écriture de ces programmes difficile pour les êtres humains, mais d'une exécution relativement facile pour les machines. L'histoire des langages de programmation est celle des efforts entrepris pour rendre ces langages plus faciles à utiliser par les êtres humains, tout en gardant la possibilité de les traduire en langage machine, et donc de les faire exécuter par des machines.

Une tendance dans l'évolution des langages de programmation est de se rapprocher des langues naturelles. Par exemple, les langages machine n'utilisent que des chiffres, mais les langages évolués également des lettres. Toutefois, les langues naturelles ne sont pas nécessairement les bons outils pour exprimer les algorithmes. Ainsi, la manière d'écrire les équations en algèbre, s'est éloignée des langues naturelles, bien avant que l'on ait des machines, simplement parce que le langage algébrique est plus facile à utiliser que les langues naturelles pour exprimer des équations et les résoudre. De même, certains langages formels sont peut-être plus faciles à utiliser que les langues naturelles pour exprimer des algorithmes. La nécessité de rendre les programmes exécutables par des machines n'est donc pas l'unique raison pour laquelle les langages de programmation sont des langages formels.

La grammaire

La première chose à définir quand on conçoit un langage formel, que ce soit un langage de programmation, un langage de description de pages web ou un langage d'interrogation de bases de données, est sa *grammaire*. La *grammaire* d'un langage est un algorithme qui indique si une chaîne de caractères appartient à ce langage ou non. Par exemple, la grammaire des instructions du langage Java indique que la chaîne de caractères `x = 1;` est une instruction, mais pas la chaîne de caractères `1 = x;`. On ne cherche pas ici à définir ce qui se passe quand on exécute cette instruction, mais uniquement si elle est grammaticalement correcte ou non.

Pour définir une grammaire, on utilise un langage qui contient :

- des symboles pour les lettres du langage, *a*, *b*, etc.
- des symboles pour des langages, c'est-à-dire des ensembles de chaînes de caractères, *A*, *B*, *L*, etc.
- les symboles `=` (égal), `|` (ou) et `ε` (chaîne vide).

On commence par un exemple simple : le langage L ne contient que deux chaînes de caractères *aab* et *abab*. La grammaire de ce langage se définit ainsi :

$L = aab \mid abab$

Un autre exemple de langage est celui qui contient toutes les chaînes de caractères qui ne sont formées que de a : la chaîne vide ε, a, aa, aaa, etc. Sa grammaire se définit ainsi :

$L = \varepsilon \mid a\,L$

Cette définition signifie qu'un élément de L est, ou bien la chaîne vide ε, ou bien la lettre a, suivie d'un autre élément de L. En effet, les chaînes de caractères, non vides, formées uniquement de a commencent toutes par un a, suivi d'une autre chaîne formée uniquement de a.

On peut, de même, définir le langage des chaînes de caractères formées d'un certain nombre de a, suivis d'un certain nombre de b. Par exemple la chaîne *aaabbbbb* fait partie de ce langage mais pas la chaîne *aaabbbab*. Pour cela, on définit d'abord un langage A qui contient toutes les chaînes qui ne sont formées que de a, puis un langage B qui contient toutes les chaînes qui ne sont formées que de b, et enfin le langage dont les éléments sont formés d'une chaîne de A suivie d'une chaîne de B.

$A = \varepsilon \mid a\,A$
$B = \varepsilon \mid b\,B$
$L = A\,B$

On peut de même définir ainsi la grammaire des instructions simples en Java. On suppose que l'on a déjà défini le langage T des types de Java, le langage V des noms de variables et le langage E des expressions. Le langage I des instructions se définit alors ainsi :

$I = T V = E ;\ I \mid V = E ; \mid I I \mid$ **`if`** `(`E`)` I **`else`** $I \mid$ **`while`** `(`E`)` $I \mid$ `{`I`}`

Une instruction est en effet :

- une déclaration : $T\,V = E ;\ I$,
- une affectation : $V = E ;$,
- une séquence : $I\,I$,
- un test : `if (`E`)` I `else` I,
- une boucle : `while (`E`)` I,
- une instruction entre accolades : `{`I`}`.

On peut également définir la grammaire du langage HTML, ou plus exactement de la partie de ce langage introduite au chapitre 8. On définit d'abord un langage qui contient les diverses lettres de l'alphabet.

> *C* = *a* | *b* | *c* | *d* | …

en excluant les symboles < et >.

Le langage formé des suites de tels symboles se définit ainsi :

> *L* = ε | *CL*

Cependant, un texte en HTML n'est pas une simple suite de tels symboles, puisqu'il peut aussi contenir des expressions comme `<b>un passage important</b>`.

On définit alors le langage des textes en HTML ainsi :

> *L* = ε | *CL* | *EL*
> *E* = *L* | <i>*L*</i> | <p>*L*</p> | <h1>*L*</h1> | <h2>*L*</h2>
> | *L*

où le langage *A*, qui reste à définir, est celui des adresses web.

Ainsi, dans le langage *E*, on introduit les symboles < et > et les différentes balises existantes, mais on assure également au passage que les balises sont utilisées correctement : toute balise ouverte est refermée par la suite et les balises sont correctement imbriquées, c'est-à-dire refermées dans l'ordre correspondant à leur ouverture. Par exemple, cette grammaire exclut une suite de caractères comme `<b><i>erreur</b></i>`.

ALLER PLUS LOIN Langages et formats

La notion de langage n'est pas très éloignée de la notion de format introduite au chapitre 9. Dans un cas comme dans l'autre, on définit un ensemble de règles qui permettent d'exprimer un texte, une image, etc. D'ailleurs, on dit parfois « le langage HTML » et parfois « le format HTML ».

On a cependant tendance à réserver le mot *langage* aux suites de symboles exprimées dans un alphabet riche et avec une grammaire relativement complexe. Ainsi le langage HTML utilise tous les symboles de l'alphabet, qui peuvent eux-mêmes être exprimés dans un format : ASCII ou latin-1, alors que le format latin-1 utilise un alphabet qui se limite aux symboles 0 et 1. De même, toutes les suites de huit bits sont bien formées en latin-1, si bien que la grammaire du format latin-1 est très simple et n'a que peu d'intérêt.

La sémantique

La grammaire d'un langage de programmation définit quand une suite de symboles est un programme bien formé dans ce langage ou non, mais pas ce qui se passe quand on exécute ce programme. Ce second volet de la définition d'un langage de programmation est appelée sa *sémantique*. Par exemple, le fait que quand on exécute l'instruction `x = 1;` on transforme l'état en mettant la valeur 1 dans la boîte de nom `x` fait partie de la sémantique, et non de la grammaire, du langage Java.

De même, la sémantique du langage HTML définit la manière dont un texte écrit en HTML s'affiche dans un navigateur.

Redéfinir la sémantique

En HTML (voir le chapitre 8), les balises `<h1>` et `</h1>` délimitent un titre et les balises `<a>` et `</a>` délimitent un lien. Concrètement, cela signifie que, dans un navigateur, un passage délimité par les balises `<h1>` et `</h1>` est écrit en gros caractères et un passage délimité par les balises `<a>` et `</a>` est écrit en bleu et souligné.

Cela dit, il est possible de modifier cette sémantique et de décider, par exemple, que les titres doivent être non seulement en grosses lettres, mais aussi en rouge et centrés et que les liens doivent être en vert et surlignés. Cela est possible car la sémantique de HTML est elle-même définie dans un langage formel : le langage CSS. Par exemple, la définition CSS suivante :

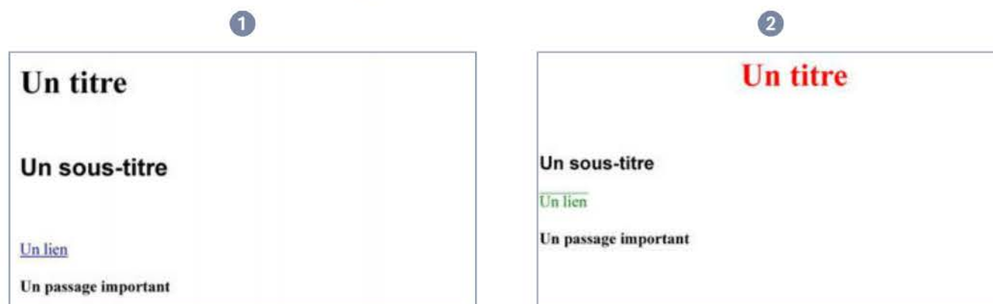
```
h1 {Font-Size: 24pt; Color: Red; Text-Align: Center;}
a {Color: Green; Text-Decoration: Overline;}
```

indique que les titres doivent être en 24 points, en rouge et centrés, et les liens en vert et surlignés.

Si on écrit cette définition dans un fichier `exemple.css` et si l'on ajoute à l'en-tête du fichier HTML, présenté au chapitre 8, la commande :

```
<link rel="stylesheet" href="example.css" type="text/css">
</link>
```

alors ce texte sera affiché dans un navigateur, non sous la forme ❶ mais sous la forme ❷.



On peut aussi définir en CSS de nouvelles balises `<oeuvre>` et `</oeuvre>` :

```
oeuvre {Font-Style: Italic}
```

Le texte :

```
<oeuvre>L'île mystérieuse</oeuvre> est à la fois la suite de
<oeuvre>Vingt mille lieues sous les mers</oeuvre> et des
<oeuvre>Enfants du capitaine Grant</oeuvre>.
```

s'affiche alors dans un navigateur :

> *L'île mystérieuse* est à la fois la suite de *Vingt mille lieues sous les mers* et des *Enfants du capitaine Grant*.

exactement comme l'aurait fait le texte :

```
<i>L'île mystérieuse</i> est à la fois la suite de <i>Vingt
mille lieues sous les mers</i> et des <i>Enfants du capitaine
Grant</i>.
```

La différence est que, si l'on décide après coup de souligner les noms des œuvres au lieu de les mettre en italique, il suffit de changer la sémantique des balises `<oeuvre>` et `</oeuvre>` :

```
oeuvre {Text-Decoration: Underline;}
```

pour que le texte s'affiche :

> L'île mystérieuse est à la fois la suite de Vingt mille lieues sous les mers et des Enfants du capitaine Grant.

les autres italiques du texte restant des italiques.

Cela permet de dissocier le fond (le texte, dans lequel on indique simplement que « L'île mystérieuse » est le titre d'une œuvre) de la forme (la sémantique du langage HTML, dans laquelle on indique que les titres des œuvres doivent être en italique ou soulignés). La mise en page des textes peut être ainsi changée *ad libitum*.



Exercice 6.1

Reprendre l'une des pages HTML écrites au chapitre 8 et lui adjoindre un fichier CSS pour modifier son aspect.

Ai-je bien compris ?

- Quelles sont les principales différences entre un langage formel et une langue naturelle ?
- Quelle est la différence entre la grammaire et la sémantique d'un langage ?
- À quoi sert le langage CSS ?

DEUXIÈME PARTIE

Informations

Dans cette deuxième partie, nous abordons l'une des problématiques centrales de l'informatique : représenter les informations que l'on veut communiquer, stocker et transformer. Nous apprenons à représenter les nombres entiers et les nombres à virgule (chapitre 7), les caractères et les textes (chapitre 8), et les images et les sons (chapitre 9). La notion de valeur booléenne, ou de bit, qui apparaît dans ces trois chapitres, nous mène naturellement à la notion de fonction booléenne (chapitre 10).

Nous apprenons ensuite à structurer de grandes quantités d'informations (chapitre 11*), à optimiser la place occupée grâce à la compression, corriger les erreurs qui peuvent se produire au moment de la transmission et du stockage de ces informations et à les protéger par le chiffrement (chapitre 12*).



XKCD

Représenter des nombres entiers et à virgule

7

Au commencement étaient le 0 et le 1, puis nous créâmes les nombres, les textes, les images et les sons.

Dans ce chapitre, nous voyons comment les nombres sont représentés dans les ordinateurs avec des 0 et des 1.

Nous introduisons la notion de base, en partant de la notation décimale que nous utilisons ordinairement pour écrire les nombres entiers. Nous passons par la base cinq puis décrivons la base deux, aussi appelée représentation binaire. Nous généralisons ensuite aux nombres relatifs en utilisant la notation en complément à deux, puis aux nombres à virgule, représentés par leur signe, leur mantisse et leur exposant.



Le livre de l'addition et de la soustraction d'après le calcul indien de **Muhammad al-Khwarizmi** (783 ? - 850 ?), qui présente la numération décimale à position et des algorithmes permettant d'effectuer les opérations sur les nombres exprimés dans ce système, a été le vecteur de la diffusion de ce système de numération dans le bassin méditerranéen. Le mot algorithme est dérivé du nom d'al-Khwarizmi et le mot algèbre (*al-jabr*) du titre d'un autre de ses livres.

Vus de l'extérieur, les ordinateurs et les programmes que nous utilisons tous les jours permettent de mémoriser, de transmettre et de transformer des nombres, des textes, des images, des sons, etc.

Pourtant, quand on les observe à une plus petite échelle, ces ordinateurs ne manipulent que des objets beaucoup plus simples : des 0 et des 1. Mémoriser, transmettre et transformer des nombres, des textes, des images ou des sons demande donc d'abord de les représenter comme des suites de 0 et de 1.

La mémoire des ordinateurs est constituée d'une multitude de petits circuits électroniques qui ne peuvent être, chacun, que dans deux états (voir le chapitre 14). Comme il fallait donner un nom à ces états, on a décidé de les appeler 0 et 1, mais on aurait pu tout aussi bien les appeler A et B, froid et chaud ou faux et vrai. Une telle valeur, 0 ou 1, s'appelle un *booléen*, un *chiffre binaire* ou encore un *bit* (*binary digit*). Un tel circuit à deux états s'appelle un *circuit mémoire un bit* et se décrit donc par le symbole 0 ou par le symbole 1.

L'état d'un circuit, composé de plusieurs de ces circuits mémoire un bit, se décrit par une suite finie de 0 et de 1, que l'on appelle un *mot*. Par exemple, le mot 100 décrit l'état d'un circuit composé de trois circuits mémoire un bit, respectivement dans l'état 1, 0 et 0.

Exercice 7.1

On imagine un ordinateur dont la mémoire est constituée de quatre circuits mémoire un bit. Quel est le nombre d'états possibles de la mémoire de cet ordinateur ? Même question pour un ordinateur dont la mémoire est constituée de dix circuits mémoire un bit. Et pour un ordinateur dont la mémoire est constituée de 34 milliards de tels circuits.

Exercice 7.2

On veut représenter chacune des sept couleurs de l'arc-en-ciel par un mot, les sept mots devant être distincts et de même longueur. Quelle est la longueur minimale de ces mots ?

Exercice 7.3

Trouvez trois informations de la vie courante qui peuvent être exprimées par un booléen.



Exercice 7.4

On considère une « box internet » avec une diode électroluminescente, éteinte ou allumée selon un motif de 0 et de 1 qui change à chaque demi-seconde. Lorsque la box est éteinte, la diode aussi, le motif est 0000000000... Lorsque la box est allumée et fonctionne, la diode est allumée en continu, le motif vaut donc 1111111111... Lorsque la box est en panne, le fournisseur d'accès souhaite que la diode clignote selon différents motifs en fonction du type de panne : pas de réseau, réseau saturé, facture non payée, etc. On parle ainsi de clignotement rapide pour le motif 0101010101.



1. Proposer deux motifs qui pourraient correspondre aux descriptions « clignotement lent » et « clignotement très lent ».
2. Comment peut-on décrire les motifs suivants, répétés indéfiniment : 0000100000 et 0101010000 ?
3. Comment faut-il procéder pour qu'un motif donné ne s'affiche que toutes les dix secondes ?
4. Dans une situation où il y aurait deux types de panne en même temps, comment pourrait-on procéder pour afficher les motifs correspondant aux deux messages d'erreurs différents ?

La représentation des entiers naturels

Depuis le Moyen Âge, on écrit les nombres entiers naturels en *notation décimale à position*. Cela signifie que, pour écrire le nombre entier naturel n, on commence par imaginer n objets, que l'on groupe par paquets de dix, puis on groupe ces paquets de dix objets en paquets de dix paquets, etc. À la fin, il reste entre zéro et neuf objets isolés, entre zéro et neuf paquets isolés de dix objets, entre zéro et neuf paquets isolés de cent, etc. Et on écrit cet entier naturel en notant, de droite à gauche, le nombre d'objets isolés, le nombre de paquets de dix, le nombre de paquets de cent, le nombre de paquets de mille, etc. Chacun de ces nombres étant compris entre zéro et neuf, seuls dix chiffres sont nécessaires : 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9. Par exemple, l'écriture 2359 exprime un entier naturel formé de 9 unités, 5 dizaines, 3 centaines et 2 milliers.

Le choix de faire des paquets de dix est peut-être dû au fait que l'on a dix doigts, mais on aurait pu tout aussi bien décider de faire des paquets de deux, de cinq, de douze, de vingt, de soixante, etc. On écrirait alors les nombres entiers naturels en notation à position en base deux, cinq, douze, vingt ou soixante. La notation décimale à position s'appelle donc aussi la notation à position en base dix.

En *notation binaire*, c'est-à-dire en notation à position en base deux, le nombre treize s'écrit <u>1101</u> : de droite à gauche, 1 unité, 0 deuzaine, 1 quatraine et 1 huitaine. L'écriture d'un entier naturel en binaire est en moyenne 3,2 fois plus longue que son écriture en base dix, mais elle ne demande d'utiliser que deux chiffres : 0 et 1.

Indiquer la base

Dans ce livre, quand une suite de chiffres exprime un nombre dans une base différente de dix, on indique la base entre parenthèses, par exemple : 1101 (en base deux). On souligne aussi parfois un mot pour indiquer qu'il exprime un nombre en base deux : <u>1101</u>. Enfin, on rassemble parfois les bits par groupe de quatre ou de huit dans les mots très longs pour qu'ils soient plus faciles à lire : <u>1111111101</u> est écrit <u>11 1111 1101</u>. Comme en base dix, ces groupes sont formés de droite à gauche.

Exercice 7.5

Un horloger excentrique a eu l'idée de fabriquer une montre sur laquelle l'heure est indiquée par 10 diodes électroluminescentes appelées 1 h, 2 h, 4 h, 8 h, 1 min, 2 min, 4 min, 8 min, 16 min et 32 min. Pour connaître l'heure, il suffit d'ajouter la valeur de toutes les diodes allumées.

Quelle heure est-il quand sont allumées les diodes 1 h, 2 h, 4 h, 1 min, 2 min, 8 min, 16 min et 32 min ? Quelles sont les diodes allumées à 5 h 55 ? Est-il possible de représenter toutes les heures ? Toutes les configurations sont-elles la représentation d'une heure ?

Comme la mémoire des ordinateurs est constituée de circuits qui ne peuvent être chacun que dans deux états, on peut utiliser chaque circuit de la mémoire pour représenter un chiffre binaire, en identifiant l'un de ces états avec le chiffre binaire 0 et l'autre avec le chiffre binaire 1 – on comprend *a posteriori* pourquoi on a choisi d'appeler ces états eux-mêmes 0 et 1. Le nombre $13 = \underline{1101}$ est donc représenté dans la mémoire d'un ordinateur par le mot 1101, c'est-à-dire par quatre circuits respectivement dans les états 1, 0, 1 et 1.

La base cinq

Pour comprendre comment transformer un entier naturel, écrit en base dix, dans une autre base, on commence par la base cinq, moins particulière que la base deux, sur laquelle on reviendra plus tard.

SAVOIR-FAIRE Trouver la représentation en base cinq d'un entier naturel donné en base dix

Pour écrire les entiers naturels en base cinq, on a besoin de cinq chiffres : 0, 1, 2, 3, 4. Quand on a n objets, on les groupe par paquets de cinq, qu'on groupe ensuite en paquets de cinq paquets, etc. Autrement dit, on fait une succession de divisions par 5, jusqu'à obtenir un quotient égal à 0.

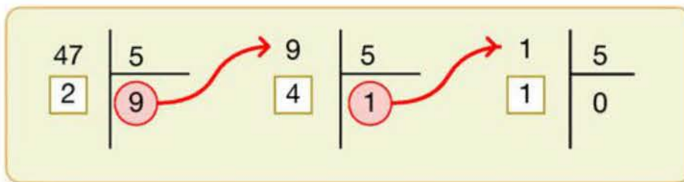
Exercice 7.6 (avec corrigé)

Trouver la représentation en base cinq de 47.

47 objets se regroupent en 9 paquets et 2 unités, puis les 9 paquets se regroupent en 1 paquet de paquets et 4 paquets.

$47 = 9 \times 5 + 2 = (1 \times 5 + 4) \times 5 + 2 = (1 \times 5^2) + (4 \times 5^1) + (2 \times 5^0)$

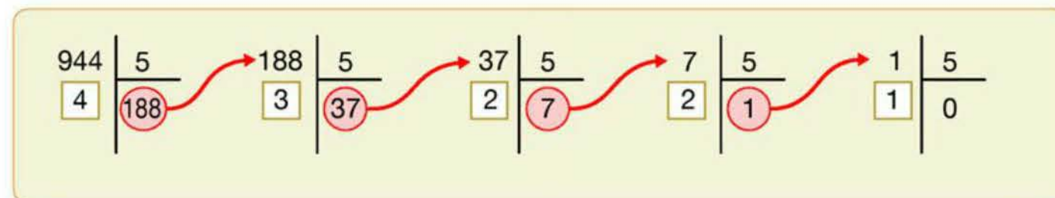
Donc 47 = 142 (en base cinq).



Exercice 7.7 (avec corrigé)

Trouver la représentation en base cinq du nombre 944.

On obtient 944 = 12234 (en base cinq).



Exercice 7.8

Trouver la représentation en base cinq du nombre 289.

SAVOIR-FAIRE Trouver la représentation en base dix d'un entier naturel donné en base cinq

Pour trouver la représentation en base dix d'un entier naturel donné en base cinq, on utilise le fait qu'en base cinq, le chiffre le plus à droite représente les unités, le précédent les paquets de 5, le précédent les paquets de $5 \times 5 = 5^2 = 25$, le précédent de $5 \times 5 \times 5 = 5^3 = 125$, et ainsi de suite.

Exercice 7.9 (avec corrigé)

Trouver la représentation en base dix du nombre 401302 (en base cinq).

401302 (en base cinq) = $(\mathbf{2} \times 5^0) + (\mathbf{0} \times 5^1) + (\mathbf{3} \times 5^2) + (\mathbf{1} \times 5^3) + (\mathbf{0} \times 5^4) + (\mathbf{4} \times 5^5)$
= 12702.

Exercice 7.10

Trouver la représentation en base dix des nombres 2341 (en base cinq) et 444 (en base cinq).

La base deux

Les nombres exprimés en base deux sont plus difficiles à lire, car il n'y a que deux chiffres, 0 et 1, mais le principe de la numération en base deux est en tout point similaire à celui de la numération en base cinq.

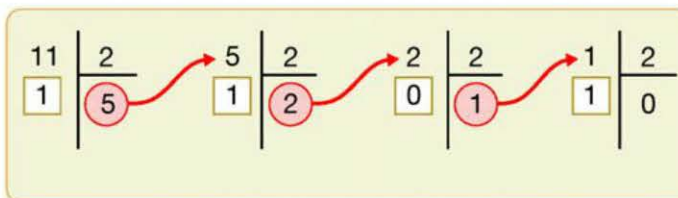
SAVOIR-FAIRE Trouver la représentation en base deux d'un entier naturel donné en base dix

Pour écrire les nombres en base deux, on a besoin de deux chiffres : 0 et 1. Quand on a *n* objets, on les groupe par paquets de deux, qu'on regroupe eux-mêmes en paquets de deux paquets, etc. Autrement dit, on fait une succession de divisions par 2, jusqu'à obtenir un quotient égal à 0.

Exercice 7.11 (avec corrigé)

Trouver la représentation en base deux du nombre 11.

On obtient 11 = 1011.



Exercice 7.12

Trouver la représentation en base deux du nombre 1000.

Exercice 7.13

Chercher sur le Web la date de la mort de Charlemagne. Trouver la représentation en base deux de ce nombre.



Exercice 7.14

Donner les représentations en base deux des nombres 1, 3, 7, 15, 31 et 63. Expliquer le résultat.

SAVOIR-FAIRE Trouver la représentation en base dix d'un entier naturel donné en base deux

Pour trouver la représentation en base dix d'un entier naturel donné en base deux, on utilise le fait qu'en base deux, le chiffre le plus à droite représente les unités, le précédent les paquets de 2, le précédent les paquets de $2 \times 2 = 2^2 = 4$, le précédent de $2 \times 2 \times 2 = 2^3 = 8$, etc.

Exercice 7.15 (avec corrigé)

Trouver la représentation en base dix du nombre 11111111.

11111111 = $(1 \times 2^0) + (1 \times 2^1) + (1 \times 2^2) + (1 \times 2^3) + (1 \times 2^4) + (1 \times 2^5) + (1 \times 2^6) + (1 \times 2^7) = 255$.

Exercice 7.16

Trouver la représentation en base dix du nombre 10010110.

Exercice 7.17

C'est en 11110010000 qu'a été démontré le théorème fondamental de l'informatique. Exprimer ce nombre en base dix.

Exercice 7.18

Montrer qu'avec un mot de n bits on peut représenter les nombres de 0 à 2^n-1.

Exercice 7.19

Pour multiplier par dix un entier naturel exprimé en base dix, il suffit d'ajouter un 0 à sa droite, par exemple, 12 × 10 = 120. Quelle est l'opération équivalente pour les entiers naturels exprimés en base deux ? Exprimer en base deux les nombres 3, 6 et 12 pour illustrer cette remarque.



Exercice 7.20

Quelle est la représentation binaire du nombre 57 ? Et celle du nombre 198 ? Soit m un mot de 8 bits, n l'entier naturel représenté en binaire par le mot m, m' le mot obtenu en remplaçant dans m chaque 0 par un 1 et chaque 1 par un 0 et n' l'entier naturel représenté en binaire par le mot m'. Exprimer n et n' comme une somme de puissances de 2, montrer que $n + n' = 255$. Montrer que la représentation binaire du nombre $255 - n$ est obtenue en remplaçant dans celle de n chaque 0 par un 1 et chaque 1 par un 0.

Les octets

Dans la mémoire des ordinateurs les circuits mémoire un bit sont souvent groupés par huit : les *octets*. On utilise souvent des nombres exprimés en notation binaire, c'est-à-dire en base deux, sur un, deux, quatre ou huit octets, soit 8, 16, 32 ou 64 bits. Ceci permet de représenter les nombres de 0 à 1111 1111 = 255 sur un octet, de 0 à 1111 1111 1111 1111 = 65 535 sur deux octets, de 0 à 1111 1111 1111 1111 1111 1111 1111 1111 = 4 294 967 295 sur quatre octets et de 0 à 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 = 18 446 744 073 709 551 615 sur huit octets.

Une base quelconque

On peut généraliser à une base quelconque les méthodes vues pour la base cinq et la base deux.

Savoir-faire Trouver la représentation en base k d'un entier naturel donné en base dix

Pour écrire les entiers naturels en base k, on a besoin de k chiffres. Quand on a n objets, on les groupe par paquets de k, qu'on regroupe à leur tour en paquets de k paquets, etc. Autrement dit, on fait une succession de divisions par k, jusqu'à obtenir un quotient égal à 0.

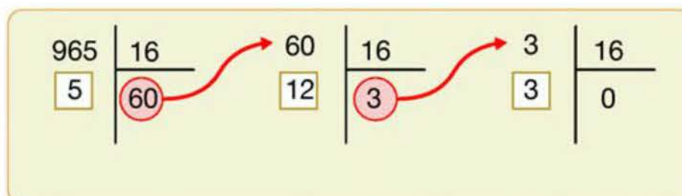
Base 16

En base seize, on a besoin de 16 chiffres : 0, 1, 2, 3, 4, 5 , 6, 7, 8, 9, puis a (dix), b (onze), c (douze), d (treize), e (quatorze) et f (quinze).

Exercice 7.21 (avec corrigé)

Trouver la représentation en base seize du nombre 965.

Réponse : 965 = 3c5 (en base seize)



Exercice 7.22

Trouver la représentation en base seize des nombres 6725 et 18 379.

Savoir-faire Trouver la représentation en base dix d'un entier naturel donné en base k

Pour trouver la représentation en base dix d'un entier naturel donné en base k, on utilise le fait qu'en base k, le chiffre le plus à droite représente les unités, le précédent les paquets de k, le précédent les paquets de $k \times k = k^2$, le précédent les paquets de $k \times k \times k = k^3$, etc.

Exercice 7.23 (avec corrigé)

Trouver la représentation en base dix du nombre 4e2c (en base seize).

4e2c (en base seize) = $(\mathbf{12} \times 16^0) + (\mathbf{2} \times 16^1) + (\mathbf{14} \times 16^2) + (\mathbf{4} \times 16^3) = 20\,012.$

Exercice 7.24

Trouver la représentation en base dix des nombres abcd (en base seize) et 281ef (en base seize).



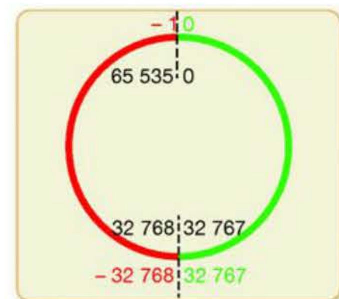
Exercice 7.25

Chercher sur le Web ce qu'est le système de numération Shadok. Est-ce un système de numération à position ? Si oui, en quelle base et avec quels chiffres ?

La représentation des entiers relatifs

Pour représenter les entiers relatifs en notation binaire, on doit étendre la représentation aux nombres négatifs. Une solution est de réserver un bit pour le signe de l'entier à représenter et d'utiliser les autres pour représenter sa valeur absolue. Ainsi, avec des mots de 16 bits, en utilisant 1 bit pour le signe et 15 bits pour la valeur absolue, on pourrait représenter les entiers relatifs de -111 1111 1111 1111 = $-(2^{15} - 1) = -32\,767$ à 111 1111 1111 1111 = $2^{15} - 1 = 32\,767$. Cependant, cette méthode a plusieurs inconvénients, l'un d'eux étant qu'il y a deux zéros, l'un positif et l'autre négatif.

On a donc préféré une autre méthode, qui consiste à représenter un entier relatif par un entier naturel. Si on utilise des mots de 16 bits, on peut représenter les entiers relatifs compris entre -32 768 et 32 767 : on représente un entier relatif x positif ou nul comme l'entier naturel x, et un entier relatif x strictement négatif comme l'entier naturel $x + 2^{16} = x + 65\,536$, qui est compris entre 32 768 et 65 535. Ainsi, les entiers naturels de 0 à 32 767 servent à représenter les entiers relatifs positifs ou nuls, en vert sur la figure, et les entiers naturels de 32 768 à 65 535 décrivent les entiers relatifs strictement négatifs, en rouge sur la figure.



L'entier relatif -1 est représenté comme l'entier naturel 65 535, c'est-à-dire par le mot 1111 1111 1111 1111.

Cette manière de représenter les entiers relatifs s'appelle la notation en *complément à deux*.

Avec des mots de seize bits, on écrit les entiers relatifs compris entre $-2^{15} = -32\,768$ et $2^{15} - 1 = 32\,767$. Plus généralement, avec des mots de n bits, on écrit les entiers relatifs compris entre -2^{n-1} et $2^{n-1} - 1$:

- un entier relatif x positif ou nul compris entre 0 et $2^{n-1} - 1$ est représenté par l'entier naturel x compris entre 0 et $2^{n-1} - 1$;
- un entier relatif x strictement négatif compris entre -2^{n-1} et -1 est représenté par l'entier naturel $x + 2^n$ compris entre 2^{n-1} et $2^n - 1$.

Exercice 7.26

Quels entiers relatifs peut-on représenter avec des mots de 8 bits ? Combien sont-ils ? Même question avec des mots de 32 bits et 64 bits.

SAVOIR-FAIRE Trouver la représentation binaire sur *n* bits d'un entier relatif donné en décimal

On a vu que :

- Si l'entier relatif x est positif ou nul, on le représente comme l'entier naturel x.
- S'il est strictement négatif, on le représente comme l'entier naturel $x + 2^n$.

Exercice 7.27 (avec corrigé)

Trouver la représentation binaire sur huit bits des entiers relatifs 0 et -128.

L'entier relatif 0 est représenté comme l'entier naturel 0 : 0000 0000.

L'entier relatif -128 est représenté comme l'entier naturel $-128 + 2^8 = -128 + 256 = 128$ *: 1000 0000.*

Exercice 7.28

Trouver la représentation binaire sur huit bits des entiers relatif 127 et -127.

SAVOIR-FAIRE Trouver la représentation décimale d'un entier relatif donné en binaire sur *n* bits

Si cet entier relatif est donné par le mot m, on commence par calculer l'entier naturel p représenté par ce mot. Si p est strictement inférieur à 2^{n-1}, c'est l'entier relatif représenté, s'il est supérieur ou égal à 2^{n-1}, l'entier relatif représenté est $p - 2^n$.

Exercice 7.29 (avec corrigé)

Trouver la représentation décimale des entiers relatifs dont la représentation binaire sur huit bits est 0000 0000 et 1000 0000.

Le mot 0000 0000 représente l'entier naturel 0 et donc l'entier relatif 0. Le mot 1000 0000 représente l'entier naturel $128 = 2^7$ *et donc l'entier relatif* $128 - 2^8 = 128 - 256 = -128$*.*

Exercice 7.30

Trouver la représentation décimale des entiers relatifs dont la représentation binaire sur huit bits est 0111 1111 et 1000 0001.

SAVOIR-FAIRE Calculer la représentation *p'* de l'opposé d'un entier relatif *x* à partir de sa représentation *p*, pour une représentation des entiers relatifs sur huit bits

Si l'entier relatif x est compris entre 0 et 127, alors il est représenté sur huit bits par l'entier naturel $p = x$ et son opposé $-x$ est représenté par l'entier naturel $p' = -x + 2^8 = -x + 256 = 256 - p$. Si l'entier relatif x est compris entre -127 et -1, alors il est représenté par l'entier naturel $p = x + 2^8 = x + 256$ et son opposé $-x$ est représenté par l'entier naturel $p' = -x = 256 - p$.

Donc, sauf quand $x = -128$, dont l'opposé n'est pas représentable sur 8 bits, si un entier relatif x est représenté par l'entier naturel p, son opposé $-x$ est représenté par l'entier naturel $p' = 256 - p = (255 - p) + 1$.
Calculer $255 - p = 1111\ 1111 - p$ est facile, puisqu'il suffit, dans la représentation binaire de p, de remplacer chaque 0 par un 1 et chaque 1 par un 0 (voir l'exercice 7.20). Il suffit ensuite d'ajouter 1 au nombre obtenu.

Exercice 7.31 (avec corrigé)

Calculer la représentation binaire sur huit bits de l'entier relatif 4, puis celle de son opposé.

L'entier relatif 4 est représenté comme l'entier naturel 4 = 0000 0100.

Pour calculer la représentation de son opposé, on remplace les 0 par des 1 et les 1 par des 0, ce qui donne 1111 1011. Puis on ajoute 1, ce qui donne 1111 1100. On peut vérifier que ce nombre est bien la représentation de l'entier relatif -4, c'est-à-dire de l'entier naturel -4 + 256 = 252.

Exercice 7.32

Calculer la représentation binaire sur huit bits de l'entier relatif -16, puis de son opposé.



Exercice 7.33

Montrer que le bit le plus à gauche vaut 1 pour les entiers relatifs strictement négatifs et 0 pour les entiers relatifs positifs ou nuls.



Exercice 7.34

On considère des entiers relatifs sur 3 bits. Dessiner le cercle rouge-vert ci-avant en plaçant les 8 nombres : 0, 1, 2, 3, -1, -2, -3, -4 à leur place. Relier les nombres opposés : 1 et -1, 2 et -2, etc. Quelle est l'interprétation géométrique de la fonction qui à chaque nombre associe son opposé ?



Exercice 7.35

Représenter les entiers relatifs 96 et 48 en binaire sur huit bits. Ajouter les deux nombres binaires obtenus en utilisant l'algorithme de l'addition binaire du chapitre 18. Quel est l'entier relatif obtenu ? Pourquoi est-il négatif ?



Exercice 7.36

Expliquer comment faire une soustraction de deux nombres binaires sur huit bits à partir du calcul de l'opposé et de l'algorithme de l'addition du chapitre 18. Calculer ainsi 15 - 7.

La représentation des nombres à virgule

Comme la notation décimale, la notation binaire permet aussi de représenter des nombres à virgule. En notation décimale, les chiffres à gauche de la virgule représentent des unités, des dizaines, des centaines, etc. et ceux à droite de la virgule, des dixièmes, des centièmes, des millièmes, etc. De même, en binaire, les chiffres à droite de la virgule représentent des demis, des quarts, des huitièmes, des seizièmes, etc. On peut ainsi représenter, par exemple, le nombre un et un quart : 1.01. Toutefois, cette manière de faire ne permet pas de représenter des nombres très grands ou très petits comme le nombre d'Avogadro ou la constante de Planck. On utilise donc une autre représentation similaire à la « notation scientifique » des calculatrices, sauf qu'elle est en base deux et non en base dix. Un nombre est représenté sous la forme $s\ m\ 2^n$ où s est le signe du nombre, n son exposant et m sa mantisse. Le signe est + ou -, l'exposant est un entier relatif et la mantisse est un nombre à virgule, compris entre 1 inclus et 2 exclu.

Par exemple, quand on utilise 64 bits pour représenter un nombre à virgule, on utilise 1 bit pour le signe, 11 bits pour l'exposant et 52 bits pour la mantisse. Le signe + est représenté par 0 et le signe - par 1. L'exposant n est un entier relatif compris entre -1 022 et 1 023 ; on le représente comme l'entier naturel n + 1 023, qui est compris entre 1 et 2 046. Les deux entiers naturels 0 et 2 047 sont réservés pour des situations exceptionnelles (+∞, -∞, NaN, etc.). La mantisse m est un nombre binaire à virgule compris entre 1 inclus et 2 exclu, comprenant 52 chiffres après la virgule. Comme cette mantisse est comprise entre 1 et 2, elle a toujours un seul chiffre avant la virgule et ce chiffre est toujours un 1 ; il est donc inutile de le représenter et on utilise les 52 bits pour représenter les 52 chiffres après la virgule.

ALLER PLUS LOIN Valeurs exceptionnelles

Ce codage prend en compte des valeurs exceptionnelles : +∞, -∞ et NaN (*not a number*) qui est une valeur indéfinie. Ces valeurs non numériques sont représentées respectivement par les mots de 64 bits suivants :

- 0 11111111111 00000000000000000000000 00000000000000000000000000000
- 1 11111111111 00000000000000000000000 00000000000000000000000000000
- 1 11111111111 11111111111111111111111 11111111111111111111111111111

Ce codage permet également de représenter des valeurs infinitésimales qui, sans être nulles, sont trop petites pour que l'on puisse calculer avec elles.

SAVOIR-FAIRE Trouver la représentation en base dix d'un nombre à virgule donné en binaire

On identifie le signe, la mantisse et l'exposant.

Exercice 7.37 (avec corrigé)

Trouver le nombre à virgule représenté par le mot
1100010001101001001111000011100000000000000000000000000000000000

Le signe est représenté par 1.
L'exposant est représenté par 10001000110. La mantisse est représentée par 1001001111000011100000000000 0000000000000000000000000.
Le signe du nombre est donc -. Le nombre <u>*100 0100 0110*</u> *est égal à 1 094 et l'exposant du nombre est* n *= 1094 - 1023 = 71. Sa mantisse est :*

```
m = 1.1001 0011 1100 0011 1000 0000 0000 0000 0000 0000 0000 0000 0000
  = 1 + 1/2 + 1/2^4 + 1/2^7 + 1/2^8 + 1/2^9 + 1/2^10 + 1/2^15 + 1/2^16 + 1/2^17
  = (2^17 + 2^16 + 2^13 + 2^10 + 2^9 + 2^8 + 2^7 + 2^2 + 2 + 1) / 2^17
  = 206727/131072.
```

Le nombre représenté est donc -206 727 / 131 072 × 2^{71} = -3.724… × 10^{21}.

Exercice 7.38

Trouver le nombre à virgule représenté par le mot :

0001000000111101001110010101100000000000000000000000000000000000.

Exercice 7.39

Comment est représenté le nombre à virgule $2^{-1\,022}$ (qui est égal à 2,225… × 10^{-308}) ?

Exercice 7.40

Comment est représenté le nombre entier 7 ? Et le nombre à virgule 7,0 ?

Exercice 7.41

À combien de décimales environ correspondent 52 chiffres binaires après la virgule ?

Exercice 7.42

Quel est le plus grand nombre à virgule que l'on peut représenter en binaire ? Quel est le plus petit nombre à virgule, donc négatif, que l'on peut représenter en binaire ? Quel est le plus petit nombre à virgule strictement positif que l'on peut représenter en binaire ?

Exercice 7.43

Quelle précision perd-on si on divise par deux un nombre à virgule avant de le remultiplier par deux ?



Exercice 7.44

À chaque multiplication de deux nombres à virgule, on arrondit le calcul en ne gardant que 52 chiffres binaires après la virgule, ce qui introduit une erreur relative de l'ordre de 2^{-52}. Quelle est la valeur de cette erreur en base dix ? Si on fait plusieurs multiplications, ces erreurs s'accumulent. Quelle est l'erreur relative d'un calcul qui est formé d'un million de multiplications, qui dure quelques millisecondes sur un ordinateur usuel ?



Exercice 7.45

On considère le programme suivant :

```
double x, y;
x = 1.0;
y = x + 1.0;
while (y - x == 1.0) {
  x = x * 2.0;
  y = x + 1.0;}
```

1. Si l'on calculait sur des nombres rationnels exacts, que se passerait-il lors de l'exécution de ce programme ?
2. Écrire ce programme et l'exécuter. Que constate-t-on ?
3. Modifier le programme de façon à déterminer au bout de combien d'exécutions du corps de la boucle il s'arrête, ainsi que la valeur de `x` à la fin de cette exécution.
4. Comment est représentée cette dernière valeur de `x` ? Et celle de `y` ?
5. Proposer une explication de ce comportement.



Exercice 7.46

On considère le programme suivant :

```
double a;
int n;
a = 0.0;
for (n = 1; n <= 10; n = n + 1) {
  a = a + 0.1;
  System.out.println(a);}
```

1. Si l'on calculait sur des nombres rationnels exacts, que se passerait-il lors de l'exécution de ce programme ?
2. Écrire ce programme et l'exécuter. Que constate-t-on ?
3. Vérifier que la représentation binaire de 0,1 est 0011111110111001100110011001100110011001100110011001100110011010.
4. Quel nombre décimal cette représentation désigne-t-elle en réalité ?
5. En déduire les représentations binaires de 0,2, 0,3 et les nombres décimaux que cette représentation désigne en réalité.
6. Expliquer le résultat obtenu.

Ai-je bien compris ?

- En quelle base représente-t-on le plus souvent les nombres en informatique ? Pourquoi ?
- Comment représente-t-on les nombres négatifs ?
- Comment représente-t-on les nombres à virgule ?

Représenter des caractères et des textes

8



Samuel Morse (1791 - 1872) est l'inventeur d'un code, dans lequel chaque lettre est exprimée par une alternance de sons brefs symbolisés par « . » et longs « – », utilisé pour télégraphier des textes. La lettre « a » y est exprimée par les sons « . – », la lettre « b » par les sons « – … », etc. Artiste peintre, Samuel Morse s'est intéressé aux télécommunications après qu'en 1925, un message lui annonçant que sa femme était malade ne lui est pas parvenu à temps. Comme nous le verrons au chapitre 12, le code morse est à références de longueurs variables, mais ce n'est pas un code préfixe.

Les lettres ? Toutes des nombres !

Dans ce chapitre, nous voyons comment sont représentés les caractères et les textes de toutes les langues du monde.

Nous expliquons pourquoi il existe plusieurs codes tels *ASCII*, *latin-1*, *latin-2*, *UTF-32*, *UTF-8*. Nous présentons ensuite les formats enrichis qui permettent de décrire la forme des caractères et des textes, comme le font les logiciels de traitement de texte.

Un exemple de format enrichi est le langage HTML.

Nous nous intéressons, dans ce chapitre, à la représentation des textes, c'est-à-dire des suites de *caractères*, éventuellement enrichies d'informations typographiques.

La représentation des caractères

Puisqu'un texte est une suite de caractères, nous commençons par nous intéresser à la représentation des caractères, c'est-à-dire entre autres choses aux lettres minuscules et majuscules, aux chiffres, aux signes de ponctuation et aux symboles mathématiques. Pour représenter ces caractères, on attribue un nombre à chacun.

Le *code ASCII*, par exemple, attribue le nombre 65 à la lettre « A », le nombre 66 à la lettre « B », le nombre 97 à la lettre « a » et le nombre 98 à la lettre « b ». Il représente 95 caractères : les 26 lettres minuscules, les 26 lettres majuscules, les 10 chiffres, les 32 symboles ! " # $ % & ' ( ) * + , – . / : ; < = > ? @ [\ ] ^ _ ` { | } ~ et 1 signe d'espace. Il représente aussi 33 autres *symboles de mise en page*, par exemple le retour chariot qui signale la fin de la ligne et le saut de page qui signale le passage à la page suivante. Le code ASCII représente donc $95 + 33 = 128 = 2^7$ caractères, par des nombres qui peuvent eux-mêmes être représentés en binaire par des mots de sept bits. Ils sont en fait représentés par des mots de huit bits, le premier étant toujours un zéro.

Le code ASCII était à l'origine conçu pour des textes écrits en anglais, comme l'indique son nom, *American Standard Code for Information Interchange*. Il n'est pas adapté pour représenter des textes écrits dans d'autres langues, même celles qui, comme le français, utilisent l'alphabet latin, car ces langues utilisent des accents, des cédilles et autres signes diacritiques. C'est pourquoi on a tout d'abord conçu une extension du code ASCII, le code *latin-1*, qui contient 191 caractères. Aux 128 caractères du code ASCII, qui sont représentés comme en ASCII, s'ajoutent les lettres « é », « É », « è », « ç », « æ », « ñ », « ö », etc. qui permettent de représenter les textes écrits dans la plupart des langues d'Europe de l'Ouest, même si, pour le français, le « œ » a été oublié. Il manque toutefois des lettres utilisées par les langues d'Europe de l'Est, si bien qu'un autre format, le code *latin-2*, a été proposé pour ces langues. Ensuite, pour représenter les textes écrits en grec, russe, chinois, japonais, coréen, etc., il a fallu proposer un format universel : *Unicode*. Unicode recense près de 110 000 caractères et associe un nom et un numéro à chacun. A priori, ce numéro se code sur 32 bits.

Cependant, Unicode existe en plusieurs déclinaisons, parmi lesquelles *UTF-32*, dans laquelle chaque caractère est ainsi exprimé sur 32 bits, et *UTF-8*, dans laquelle les caractères les plus courants sont exprimés sur 8 bits et les moins courants sur 16, 32 ou 64 bits, utilisant une idée discutée en détail au chapitre 12 à propos de la notion de compression.

Le format UTF-8 a vocation à devenir le standard, mais il ne l'est pas encore : malgré les efforts des comités de normalisation, l'humanité n'a pas encore réussi à se doter d'un format universellement accepté, si bien qu'il est parfois nécessaire de traduire un texte d'UTF-8 en latin-1 ou de latin-2 en UTF-8. Quand cette traduction n'est pas bien faite, les caractères accentués sont remplacés par des caractères bizarres.

Cependant, tous ces formats reposent sur une même idée : associer un nombre, c'est-à-dire un mot binaire, à chaque caractère. Tous ces formats sont accessibles sur le Web.

La représentation des textes simples

Un texte étant une suite de caractères, on peut le représenter en écrivant les caractères les uns après les autres.

SAVOIR-FAIRE Trouver la représentation en ASCII binaire d'un texte

En utilisant une table, on cherche le code ASCII de chaque caractère. Puis on traduit chacun de ces nombres en représentation binaire.

Exercice 8.1 (avec corrigé)

Trouver la représentation binaire en ASCII du texte « Je pense, donc je suis. »

On cherche la table des codes ASCII sur le Web de manière à traduire le texte, caractère par caractère : 74, 101, 32, 112, 101, 110, 115, 101, 44, 32, 100, 111, 110, 99, 32, 106, 101, 32, 115, 117, 105, 115, 46. On exprime ensuite chacun de ces nombres en binaire sur huit bits :
01001010 01100101 00100000 01110000 01100101 01101110 01110011 01100101 00101100 00100000 01100100 01101111 01101110 01100011 00100000 01101010 01100101 00100000 01110011 01110101 01101001 01110011 00101110.

Exercice 8.2

Trouver la représentation binaire en ASCII du texte « Cet exercice est un peu fastidieux. »

Savoir-faire Décoder un texte représenté en ASCII binaire

On découpe la suite de bits en octets, on traduit chaque octet en décimal, puis on cherche en utilisant une table, le caractère exprimé par chacun de ces nombres.

Exercice 8.3 (avec corrigé)

Trouver le texte représenté en ASCII binaire par la suite de bits 01000011001001110110010101110011011101000010000001100110011000010 110001101101001011011000 1100101.

On commence par découper la suite de bits en octets : 01000011 00100111 01100101 01110011 01110100 00100000 01100110 01100001 01100011 01101001 01101100 01100101. Chaque octet représente un nombre entier : 67, 39, 101, 115, 116, 32, 102, 97, 99, 105, 108, 101. On cherche ensuite la table des codes ASCII en ligne de manière à traduire chacun de ces nombres en une lettre : « C'est facile ».

Exercice 8.4

Trouver le texte représenté en ASCII binaire par la suite de bits 0011000001110100011001010111010001110100000110001.

Exercice 8.5

Traduire en ASCII binaire votre phrase préférée, par exemple : « Le commencement de toutes les sciences, c'est l'étonnement. » en oubliant les accents. Traduire ensuite cette phrase en UTF-8 avec les accents.

Exercice 8.6

Traduire une phrase en ASCII binaire, puis la passer à son voisin qui la décode.

Exercice 8.7

On suppose que les seize lettres qui suivent sont codées ainsi :

ء	آ	أ	ؤ	إ	ئ	ا	ث	د	ش	ف	ك	ل	ن	و	ي
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Décoder le message suivant : 1011 1100 1001 1111 0000 1010 1111 0111 1101 0110 0101 1111 0110 1100 1011 1110 1000, puis en se faisant éventuellement aider d'une personne qui lit l'arabe ou en utilisant le mécanisme de traduction de Google, déterminer s'il correspond à la phrase « tout en code binaire » ou « les lettres deviennent des nombres ».



Exercice 8.8

On considère l'alphabet de 32 signes constitué des 26 lettres de l'alphabet, de l'espace et de cinq symboles de ponctuation : la virgule, le point, le point-virgule, le point d'interrogation et le point d'exclamation. On représente les caractères de cet alphabet par un code binaire de cinq bits présenté dans le tableau suivant. Sur la dernière ligne, on a fait figurer la traduction décimale de chaque code binaire.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z		,	.	;	?	!
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**

1. Quel est le troisième bit du code de la lettre « P » ?
2. Décoder le message suivant : 000010111001101010010111010100 10001.
3. Écrire son prénom avec ce code.
4. Quels sont les caractères qui commencent par deux 1 ?
5. Pourquoi n'y a t-il pas de caractères en minuscule dans cet alphabet ?
6. On considère l'opération qui consiste à transformer les 0 en 1 et vice-versa dans chaque caractère. Recopier et remplir le tableau ci-après qui, à chaque caractère, associe le caractère obtenu par cette transformation.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z		,	.	;	?	!
!	?																														

7. On considère l'opération qui consiste à échanger le premier bit avec le dernier, et le deuxième bit avec l'avant-dernier ; recopier et remplir le tableau ci-après qui, à chaque caractère, associe le caractère obtenu par cette transformation.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z		,	.	;	?	!
A	Q																														

Attention **Utiliser un code standard**

Pour les exercices 8.7 et 8.8, nous avons inventé un code valable exclusivement pour la durée de l'exercice. Pour partager des informations, il ne faut jamais faire cela, mais utiliser un code standard, connu de tous. Sauf, bien sûr, si on veut garder le texte secret (voir le chapitre 12).

Exercice 8.9

Selon Pierre Lecomte du Noüy, 01001100 01100101 00100000 01100010 01110101 01110100 00100000 01100100 01100101 00100000 01101100 01100001 00100000 01110011 01100011 01101001 01100101 01101110 01100011 01100101 00100000 01100101 01110011 01110100 00100000 01100100 01100101 00100000 01110000 01110010 01100101 01110110 01101111 01101001 01110010 00101100 00100000 00100000 01101110 01101111 01101110 00100000 01100100 01100101 00100000 01100011 01101111 01101101 01110000 01110010 01100101 01101110 01100100 01110010 01100101. Êtes-vous d'accord avec lui ?

La représentation des textes enrichis

Les textes en ASCII ou en Unicode sont simplement des suites de caractères. Les *éditeurs de texte* sont les logiciels qui manipulent ces suites de caractères. Toutefois, quand on écrit un texte, on peut souhaiter lui donner une forme spéciale, plus jolie, plus lisible, comme le fait un imprimeur. On peut jouer sur la police de caractères – Times, Courier, etc. –, sur la taille des caractères – 11 points, 12 points, etc. –, sur leur forme – romain, italique, etc. –, leur graisse – maigre, gras, etc. On peut aussi souhaiter découper un texte en chapitres et mettre en valeur les titres des chapitres, etc. Or, les seules caractéristiques que l'on puisse exprimer avec un code comme l'ASCII, par exemple, sont la casse d'une lettre – minuscule ou majuscule – et le découpage en paragraphes, grâce au symbole retour chariot. Les *traitements de texte* sont les logiciels qui permettent ces mises en pages plus élaborées.

Ceci a amené à enrichir ces formats, de manière à :

1. *qualifier* certaines parties du texte, par exemple en mettant certaines parties en gras ou en italique,
2. structurer le texte en *divisions* : un texte n'est pas uniquement une suite de paragraphes, mais est hiérarchisé en parties, chapitres, sections, sous-sections…,
3. présenter certaines informations sous forme de listes et de tables,
4. permettre de faire *référence* à d'autres textes,
5. donner des informations sur le texte : son titre, son ou ses auteur(s), sa date de création, sa langue, des mots-clés utilisés pour le rechercher parmi plusieurs textes, etc. Ces informations **sur** le texte, et non **du** texte, sont appelées des *méta-données*.

Toutes ces considérations sont, bien entendu, valables aussi bien pour les textes manuscrits ou imprimés que pour les textes traités par les ordinateurs.

L'un de ces formats enrichis, qui est utilisé en particulier pour écrire des pages web est appelé le format HTML. En HTML, pour mettre un passage en gras, on le délimite par les *balises* `<b>` et `</b>` et pour le mettre en italique, on le délimite par les balises `<i>` et `</i>`. Ainsi le texte :

```
Ma <i>première</i> page <b>web</b>
```

s'affiche dans un navigateur :

> Ma *première* page **web**

Comme les parenthèses, les balises vont par deux : on ouvre le passage à mettre en gras avec la balise `<b>` et on le ferme avec la balise `</b>`.

Une division du texte est délimitée par les balises `<div>` et `</div>`, ainsi le texte :

```
<div>Ma première page web
<div> comporte une première sous-division pour dire « Bonjour tout le
monde ! »</div>
<div> et une seconde qui finit par « À bientôt ! »</div>
</div>
```

s'affichera dans le navigateur en rendant ces divisions explicites.

Comme les parenthèses, les balises peuvent s'emboîter les unes dans les autres, mais pas se chevaucher.

On indique qu'un passage est un titre en le délimitant par les balises `<h1>` et `</h1>` et que c'est un sous-titre en le délimitant par les balises `<h2>` et `</h2>`.

De même, les autres structurations du texte comme les énumérations ou les tableaux sont exprimées par d'autres balises.

Quand on écrit un texte, il est fréquent de mentionner d'autres textes : par exemple, de parler dans une lettre d'un livre que l'on a lu. Dans le cas d'un texte manuscrit ou imprimé, on donne en général une référence du texte cité, par exemple le titre du livre et son auteur, afin que le lecteur puisse s'y référer s'il le souhaite. Quand on veut exprimer, dans une page web, une référence à une autre page, on peut faire mieux que simplement indiquer l'adresse de la page web en question (par exemple l'adresse http://fr.wikipedia.org/wiki/Hypertext_Markup_Language) ; on peut changer l'apparence du passage où l'on fait la référence, pour indiquer au lecteur que s'il clique sur ce passage, le navigateur affichera la page demandée. On utilise pour cela les balises `<a>` et `</a>` : on encadre la partie du texte à qualifier par ces deux balises et on indique à l'intérieur de la balise `<a>` l'adresse de la page référencée. Par exemple le texte :

```
Pour les détails sur le langage HTML, on pourra consulter <a href =
"http://fr.wikipedia.org/wiki/Hypertext_Markup_Language">la page
<i>Hypertext Markup Language de Wikipédia</i></a>.
```

qui affiche dans un navigateur :

> Pour les détails sur le langage HTML, on pourra consulter la page *Hypertext MarkupLanguage de Wikipédia*.

Si l'on clique sur le passage en bleu et souligné, le navigateur affiche la page dont l'adresse est http://fr.wikipedia.org/wiki/Hypertext_Markup_Language. Un tel passage sur lequel on peut cliquer pour accéder à une autre page s'appelle un *lien*, et un texte qui contient au moins un lien est un *hypertexte*.

Les balises <body> et </body> délimitent le texte à afficher dans le navigateur. On indique avant ces informations les méta-données relatives à la page : son titre, le format utilisé pour les lettres accentuées, etc.

Voici, au bout du compte, un exemple de texte au format HTML :

```
<html>
<head>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8">
  </meta>
  <title>Un exemple</title>
</head>

<body>
  <h1>Un titre</h1>
  <h2>Un sous-titre</h2>
  <div><a href="http://www.wikipedia.org/">Un lien</a></div>
  <div><b>Un passage important</b></div>
</body>
</html>
```

L'en-tête situé entre les deux balises <head> et </head> indique d'une part que le texte est exprimé en UTF-8, c'est l'objet de la ligne :

```
<meta http-equiv="content-type" content="text/html; charset=UTF-8">
</meta>
```

et d'autre part que le titre de la page est `Un exemple`.

Le contenu est situé entre les balises <body> et </body>. On y retrouve les balises , , <i>, </i>, <h1>, </h1>, <h2>, </h2>, <div>, </div>, <a> et que l'on a décrites. Dans un navigateur, le texte s'affiche de la manière ci-contre.

Un titre

Un sous-titre

Un lien

Un passage important

SAVOIR-FAIRE Écrire une page en HTML

Écrire le texte contenu dans cette page. Structurer ce texte en divisions. Identifier les titres de parties, les passages à mettre en gras, en italique, etc. et les références vers d'autres pages. Ajouter les balises <body> et </body> autour du corps du texte, l'en-tête qui contient les méta-données et terminer avec les balises <html> et </html>.

Exercice 8.10 (avec corrigé)

Écrire une page HTML qui présente les différents projets informatiques des élèves d'une classe.

```
<html>
<head>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8"></meta>
  <title>Projets Ada Lovelace</title>
</head>

<body>
  <h1>Les projets de la classe de TS1 du Lycée <a href = "http://www.adalovelace.fr">Ada Lovelace</a></h1>
  <div>
   <a href="http://www.adalovelace.fr/informatique/ts1/projets/backgammon/index.html">Un programme qui <b>joue
aux Backgammon</b></a>
  </div>
  <div>
   <a href="http://www.adalovelace.fr/informatique/ts1/projets/compression/index.html">Un programme qui
<b>compresse des images</b></a>
  </div>
  <div><a href="http://www.adalovelace.fr/informatique/ts1/projets/montecarlo/index.html">Un programme qui
<b>calcule des intégrales</b> sans peine</a>
  </div>
</body>
</html>
```

Exercice 8.11

Écrire une page HTML qui présente la liste des concerts et des spectacles présentés dans un théâtre.

Exercice 8.12

Changer le texte HTML `Ma <i>première</i> page web` pour que le mot « première » apparaisse non en italique, mais en gras.

Exercice 8.13

Ce texte HTML est incorrect. Comment le corriger ?

```
Il faut <i>comprendre/i> le codage des objets numériques pour les
maîtriser.
```

Exercice 8.14

Dans ce texte, vers quel site web pointe le lien ?

```
Votre compte bancaire présente une anomalie. Cliquer <a href="http://grosse-arnaque.com">ici</a> pour avoir de l'aide.
```

Comment ce texte s’affiche-t-il dans un navigateur ? Quel est l’intérêt de regarder le source HTML de cette page avant de cliquer ?

Exercice 8.15

Donner le source HTML du texte suivant sachant que le texte en bleu et souligné est un lien vers la page http://www.monlivre.fr/page2 :

> On pourra consulter la page suivante.

Ai-je bien compris ?

- Comment représente-t-on un caractère ?
- Quelle est la différence entre le code ASCII et le format Unicode ?
- Quelle est la différence entre le format Unicode et le format HTML ?

Représenter des images et des sons

9



Claude Shannon (1916 - 2001), a montré en 1949, en s'appuyant sur des travaux antérieurs de Harry Nyquist, que la fréquence d'échantillonnage d'un son, et plus généralement d'un signal, doit être au moins le double de la fréquence maximale contenue dans ce son, pour que le son puisse être restitué à partir de l'échantillon. Il a également montré comment décrire les circuits électroniques par des fonctions booléennes (voir le chapitre 13) et comment exprimer toutes les fonctions booléennes et arithmétiques à l'aide des fonctions booléennes élémentaires (voir le chapitre 10).

Pour décrire une image, commence par comprendre comment ton œil la voit.

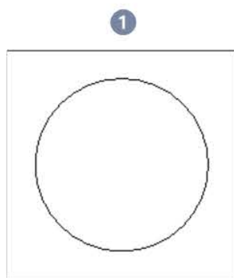
C'est encore avec des 0 et des 1 que l'on représente les images et les sons, mais en grand nombre.

Pour décrire une image, on peut utiliser une représentation vectorielle en décrivant des formes géométriques : un cercle, une droite, etc., ou une représentation bitmap en quadrillant l'image et en décrivant chaque case (*pixel*). Noir et blanc, niveaux de gris ou couleurs sont décrits par différents codages.

Les sons aussi sont échantillonnés en découpant le temps durant lequel le son est émis.

Les images et les sons sont de très longues suites de 0 et de 1, nous introduisons dans ce chapitre les unités pour mesurer la taille des fichiers.

La représentation des images

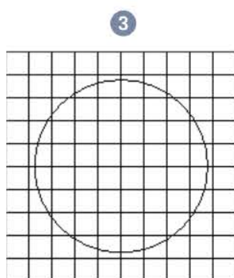


Pour décrire l'image 1, une possibilité est de dire : « cette image est formée d'un cercle ». On peut même être plus précis et indiquer les coordonnées du centre du cercle et son rayon. Et, à partir de cette description, n'importe qui pourrait reconstituer le dessin.



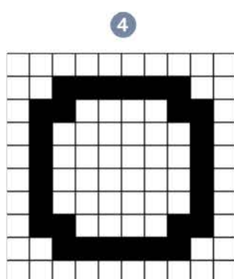
On peut donc représenter cette image par trois nombres : deux pour les coordonnées du centre et un pour le rayon. Une image formée de plusieurs cercles serait de même décrite par trois nombres pour chacun d'eux. On peut représenter d'une manière similaire un dessin formé de cercles et de rectangles, en représentant chaque figure par une lettre, « c » pour un cercle, « r » pour un rectangle, suivi d'une suite de nombres qui définissent les paramètres de la figure. On parle alors de représentation *symbolique*, ou parfois de représentation *vectorielle*, d'une image.

Néanmoins, cette méthode est peu pratique pour représenter l'image 2.



Une autre méthode, qui a l'avantage de pouvoir être utilisée sur n'importe quelle image, consiste à superposer un quadrillage à l'image 3.

Chacune des cases de ce quadrillage s'appelle un *pixel* (*picture element*). On noircit ensuite les pixels qui contiennent une portion de trait (4).



Puis, il suffit d'indiquer la couleur de chacun des pixels, en les lisant de gauche à droite et de haut en bas, comme un texte. Ce dessin se décrit donc par une suite de mots « blanc » ou « noir ». Comme seuls les mots « noir » ou « blanc » sont utilisés, on peut être plus économe et remplacer chacun de ces mots par un bit, par exemple 1 pour « noir » et 0 pour « blanc ».

Le dessin ci-avant, avec une grille de 10 × 10, se décrit alors par la suite de 100 bits suivante :

```
0000000000001111110001100001100100000010010000001001000000100100000010
011000011000111111000000000000
```

Cette description est assez approximative, mais on peut la rendre plus précise en utilisant un quadrillage, non plus de 10 × 10 pixels, mais de 100 × 100 pixels. À partir de quelques millions de pixels, notre œil n'est plus capable de faire la différence entre les deux images. Cette manière de représenter une image sous la forme d'une suite de pixels, chacun exprimés sur un bit, s'appelle une *bitmap*. C'est une méthode approximative, mais universelle : n'importe quelle image en noir et blanc peut se décrire ainsi.

La notion de format

Nous verrons au chapitre 11 que quand on stocke un texte, une image, un son, etc. par exemple sur un disque, on le stocke dans un *fichier*. Pour le moment, nous avons juste besoin de savoir qu'un fichier est un mot, c'est-à-dire une suite de 0 et de 1, auquel on a attribué un nom.

Si on trouve un fichier qui contient la suite de bits

0000000000001111110001100001100100000010010000001001000000
0100100000010011000011000111111000000000000

il n'est pas *a priori* évident que cette suite exprime une image de 10 × 10 pixels. Il pourrait aussi s'agir par exemple d'un texte en ASCII, ou de la représentation d'un nombre à virgule. Pour cette raison, au lieu d'appeler le fichier simplement `cercle`, on l'appelle `cercle.pbm`. Cette extension `pbm` du nom indique que ce fichier est exprimé dans le format PBM (*Portable BitMap*). Ce format est l'un des plus simples pour exprimer des images. Un fichier au format PBM est un fichier en ASCII qui se compose comme suit :

- les caractères `P1`, suivis d'un retour à la ligne ou d'un espace,
- la largeur de l'image, en base 10, suivie d'un retour à la ligne ou d'un espace,
- la hauteur de l'image, en base 10, suivie d'un retour à la ligne ou d'un espace,
- la liste des pixels, ligne par ligne, de haut en bas et de gauche à droite – les retours à la ligne et les espaces sont ignorés dans cette partie.

En outre, aucune ligne ne doit dépasser 70 caractères et toutes les lignes commençant par le caractère `#` sont des commentaires ignorés (voir le chapitre 3). Ainsi, le code ci-contre est un fichier au format PBM. Bien entendu, le format PBM n'est que l'un des multiples formats de fichiers d'images. D'autres exemples sont PGM, PPM, PNG, JPEG, GIF, PS, PICT, TIFF, etc.

```
P1
# Mon premier fichier PBM : cercle
10 10
0000000000
0011111100
0110000110
0100000010
0100000010
0100000010
0100000010
0110000110
0011111100
0000000000
```

Savoir-faire Identifier quelques formats d'images

Les différents formats qui permettent d'exprimer des images se distinguent les uns des autres par :

- le type d'image : noir et blanc, en niveaux de gris, en couleurs, etc. ;
- la manière d'exprimer ces images : sous forme vectorielle ou de bitmap ;
- le fait que ces images soient compressées ou non ;
- le fait que le format soit public ou secret : certains formats sont publics, d'autres sont gardés secrets par leurs concepteurs, de manière à contraindre les utilisateurs à utiliser leurs logiciels pour traiter ces images ;
- le fait que ces formats soient propriétaires ou libres : pour utiliser certains formats, il est nécessaire de payer des droits à leurs concepteurs, alors que pour d'autres non.

Exercice 9.1 (avec corrigé)

Chercher sur le Web les caractéristiques du format PBM.

C'est un format noir et blanc, bitmap, non compressé, public et libre.

Exercice 9.2

Chercher sur le Web les caractéristiques des formats GIF et PNG.

La représentation des images en niveaux de gris et en couleurs

Certaines images, par exemple les photos en noir et blanc, utilisent, en plus du noir et du blanc, diverses nuances de gris. On les appelle les images *en niveaux de gris*. Un format, parmi d'autres, pour exprimer ces images est le format PGM (*Portable GreyMap*). Pour exprimer une image dans le format PGM, on choisit une valeur maximale, par exemple 255, pour exprimer les niveaux de gris et on associe à chaque pixel un nombre compris entre 0 et 255, 0 indiquant que le pixel est noir et 255 qu'il est blanc. Les valeurs de 1 à 254 expriment différentes teintes de gris, de la plus foncée à la plus claire. Un fichier au format PGM, ressemble beaucoup à un fichier au format PBM, c'est un fichier en ASCII qui se compose comme suit :

- les caractères `P2`, suivis d'un retour à la ligne ou d'un espace,
- la largeur de l'image, suivie d'un retour à la ligne ou d'un espace,
- la hauteur de l'image, suivie d'un retour à la ligne ou d'un espace,

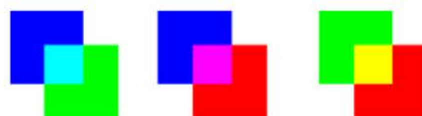
- la valeur maximale utilisée pour exprimer les niveaux de gris, suivie d'un retour à la ligne ou d'un espace,
- la liste des couleurs des pixels, ligne par ligne, de haut en bas et de gauche à droite, séparées par des retours à la ligne ou des espaces.

Comme en PBM, aucune ligne ne doit dépasser 70 caractères et toutes les lignes commençant par le caractère # sont ignorées.

Pour comprendre comment représenter les images en couleurs, il faut d'abord s'intéresser à la manière dont notre œil les perçoit. Notre œil contient des cellules, les *cônes*, qui sont sensibles à la couleur, c'est-à-dire à la longueur d'onde de la lumière qu'ils reçoivent. Ces cônes sont de trois sortes, dont le maximum de sensibilité est respectivement dans le rouge (560 nm), le vert (530 nm) et le bleu (424 nm). Quand notre œil reçoit une lumière monochrome émise par une ampoule jaune, les cônes sensibles au rouge et au vert réagissent beaucoup et ceux sensibles au bleu un tout petit peu, exactement comme s'il recevait un mélange de lumières émises par deux ampoules rouge et verte. Ainsi, en mélangeant de la lumière produite par une ampoule rouge et une ampoule verte, on peut donner à l'œil la même sensation que s'il recevait une lumière jaune. Plus généralement, quelle que soit la lumière qu'il reçoit, notre œil ne communique à notre cerveau qu'une information partielle : l'intensité de la réaction des cônes sensibles au rouge, au vert et au bleu. Et deux lumières qui stimulent ces trois types de cônes de manière identique sont indiscernables pour l'œil.

Ainsi, sur l'écran d'un ordinateur, chaque pixel est composé non pas d'une, mais de trois sources de lumière, rouge, verte et bleue ; en faisant varier l'intensité de chacune de ces sources, on peut simuler n'importe quelle couleur.

Par exemple, en mélangeant de la lumière verte et de la lumière bleue on obtient de la lumière cyan. En mélangeant de la lumière rouge et de la lumière bleue on obtient de la lumière magenta. Et en mélangeant de la lumière rouge et de la lumière verte on obtient de la lumière jaune. « Cyan » est le nom savant d'un bleu turquoise et « magenta » celui d'un rouge tirant un peu sur le violet.



Ce procédé de représentation des couleurs dépend donc à la fois de la physique de la lumière et de la biologie de la vision : si, comme certains oiseaux, nous avions quatre types de cônes, la physique de la lumière serait la même, mais nous devrions cependant concevoir des écrans dans lesquels chaque pixel contient, non pas trois, mais quatre sources lumineuses.

Un format, parmi d'autres, pour exprimer ces images est le format PPM (*Portable PixMap*). Pour exprimer une image dans ce format, on choisit une valeur maximale, par exemple 255, pour exprimer l'intensité des couleurs et on associe à chaque pixel trois nombres, l'intensité en rouge,

en vert et en bleu, chaque nombre étant compris entre 0 et 255. Un fichier au format PPM, ressemble beaucoup à un fichier au format PBM ou PGM ; c'est un fichier ASCII qui se compose comme suit :

- les caractères `P3`, suivis d'un retour à la ligne ou d'un espace,
- la largeur de l'image, suivie d'un retour à la ligne ou d'un espace,
- la hauteur de l'image, suivie d'un retour à la ligne ou d'un espace,
- la valeur maximale utilisée pour exprimer l'intensité des couleurs,
- la liste des valeurs des couleurs, trois par pixel, dans l'ordre rouge, vert, bleu, ligne par ligne, de haut en bas et de gauche à droite, séparées par des retours à la ligne ou des espaces.

Comme en PBM et en PGM, aucune ligne ne doit dépasser 70 caractères et toutes les lignes commençant par le caractère `#` sont ignorées.

```
P3
# Mon premier fichier PPM : orange
100 100
255
237 127 16
237 127 16
237 127 16
237 127 16
…
```

Par exemple, en mélangeant du rouge et du vert en quantités égales, on obtient du jaune. En augmentant la quantité de rouge, par exemple rouge = 237, vert = 127, bleu = 16, on obtient du orange. On peut alors écrire un fichier PPM qui représente un carré orange de 100 pixels sur 100 pixels, le triplet `237 127 16` devant être répété dix mille fois, puisque l'image est formée de dix mille pixels (voir ci-contre).

Cet exemple aide à comprendre pourquoi il y a des formats d'images plus complexes que le format PPM : il y a de nombreux moyens d'éviter de recopier dix mille fois le même triplet de nombres. C'est ce que l'on appelle *compresser* un fichier (voir le chapitre 12).

SAVOIR-FAIRE Numériser une image sous forme d'un fichier

1 Identifier le type d'image à numériser (noir et blanc, en niveaux de gris ou en couleurs) et choisir un format approprié.

2 Identifier la valeur de chaque pixel.

3 Identifier les autres informations que contient le fichier : taille de l'image, commentaires, etc. et la manière dont ces informations sont exprimées dans ce format.

Exercice 9.3 (avec corrigé)

1 Lequel des formats PBM, PGM et PPM est adapté pour représenter un carré noir de 10 pixels sur 10 pixels ?

2 Même question pour un carré rouge de même taille.

3 Comparer les tailles des fichiers obtenus.

1 *Pour un carré noir, le format PBM suffit, puisqu'il n'y a ni niveaux de gris ni couleurs, et le fichier est le suivant (voir ci-contre* 1*).*

2 *Pour un carré rouge, il faut obligatoirement recourir au format PPM, seul capable de représenter de la couleur. Le rouge se représente par les trois nombres* `255 0 0` *: intensité maximale pour le rouge et nulle pour le vert et le bleu. On obtient le fichier* 2.

1
```
P1
# Un carré noir
10 10
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
```

3 *Le fichier représentant la seconde image est significativement plus gros que celui représentant la première, puisqu'il faut indiquer trois octets par pixel, au lieu d'un bit dans le cas du fichier PBM.*

Exercice 9.4

Écrire les fichiers de l'exercice 9.3 dans un éditeur de texte et les ouvrir avec un logiciel de traitement d'images, par exemple Gimp. Attention, certains dessins sont petits, il peut être nécessaire de zoomer pour bien les voir.

2
```
P3
# Un carré rouge
10 10
255
255 0 0
255 0 0
255 0 0
...
(100 lignes identiques)
```



Exercice 9.5

Rechercher des informations sur la structure des fichiers GIF.

1. Combien de bits occupe la représentation d'un pixel dans ce format ?
2. Quelle information particulièrement importante pour l'affichage de l'image le fichier doit-il contenir et qui n'était pas présente dans les formats PBM, PGM et PPM ?



Depuis Georges Seurat et Paul Signac, à la fin du XIXe siècle, les artistes exploitent cette possibilité surprenante de suggérer des images, avec des taches monochromes. Ici un Mario en carrelage, photographié rue au Plâtre, à Paris, en juillet 2011.



Logo de DansTonChat.com en pixel art

ALLER PLUS LOIN La synthèse soustractive

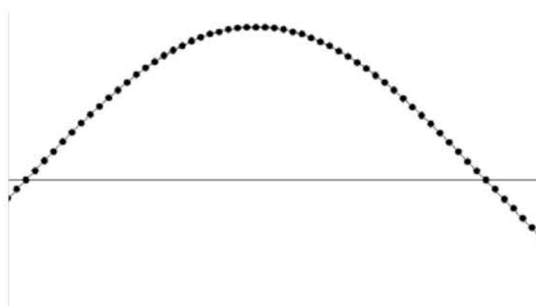
Les imprimantes également simulent toutes les couleurs en mélangeant trois types d'encre. Pourtant ces trois encres ne sont pas rouge, vert et bleu, mais cyan, magenta et jaune. Cela est dû au fait que, contrairement à un écran qui émet de la lumière, une feuille de papier ne fait que recevoir de la lumière blanche (c'est-à-dire un mélange de lumières de toutes les couleurs), absorber certaines couleurs et refléter les autres. L'encre rouge, par exemple, absorbe le vert et le bleu et reflète le rouge. De même, l'encre verte absorbe le rouge et le bleu et reflète le vert. Si on mélange de l'encre rouge et de l'encre verte, on obtient une encre qui absorbe le rouge, le vert et le bleu et qui ne reflète rien : une encre noire, et non jaune. C'est pour cela qu'en *synthèse soustractive*, on doit utiliser trois encres telles que :

- La première absorbe le rouge et reflète le vert et le bleu : l'encre cyan.
- La deuxième absorbe le vert et reflète le rouge et le bleu : l'encre magenta.
- La troisième absorbe le bleu et reflète le rouge et le vert : l'encre jaune.

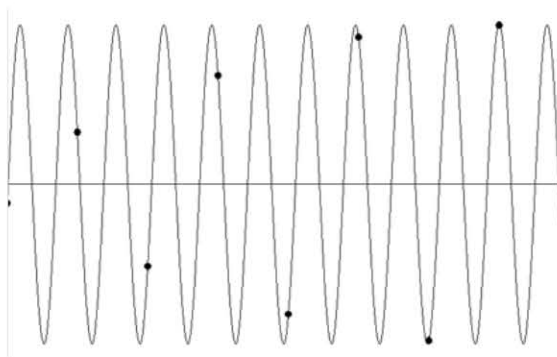
Ce même principe de synthèse soustractive est celui qu'emploient les peintres pour obtenir toutes les couleurs en mélangeant, sur leur palette, trois couleurs primaires.

La représentation des sons

Un son est une variation de la pression de l'air au cours du temps. Une manière simple de représenter un son consiste à l'*échantillonner*, c'est-à-dire mesurer la pression à intervalles réguliers et le représenter comme la suite des mesures obtenues. L'échantillonnage d'un son est une méthode assez similaire au découpage d'une image en pixels, sauf que le découpage s'effectue, non dans l'espace, mais dans le temps. Par exemple, si on échantillonne à 44 000 Hz, c'est-à-dire en faisant 44 000 mesures par seconde, une sinusoïde de fréquence 440 Hz, on fait cent mesures par période et on obtient l'échantillon suivant :



En revanche, si l'on échantillonne à 300 Hz cette même sinusoïde, on fait une mesure toutes les périodes et demie environ, et on obtient l'échantillon suivant :



On comprend qu'il est possible de reconstituer la sinusoïde dans le premier cas, mais pas dans le second. De manière plus générale, on montre que quand on échantillonne un son, il faut, pour que la reconstitution soit possible, une fréquence d'échantillonnage au moins double de la fréquence la plus élevée contenue dans ce son.

ALLER PLUS LOIN La représentation des sons et la notation musicale

Cette manière de représenter les sons par échantillonnage se distingue de la notation musicale utilisée depuis le XIIIe siècle, qui permet de représenter la durée, la fréquence et l'intensité des notes de musique, chacune représentée par un symbole sur une portée. Des systèmes intermédiaires entre l'échantillonnage et la notation musicale existent aussi, comme le format MIDI (*Musical Instrument Digital Interface*) utilisé pour représenter les sons produits par les instruments de musique électroniques.

En général, on échantillonne les sons à 44 000 Hz, car le son sinusoïdal le plus aigu que notre oreille peut entendre est de 22 000 Hz environ, ce qui implique que notre oreille ne peut pas distinguer deux sons qui donnent le même échantillon à 44 000 Hz. Cette fréquence est relativement élevée ; c'est pourquoi il faut plusieurs millions de bits pour représenter une minute de son et les fichiers audio sont souvent compressés (voir le chapitre 12).

Cette méthode de représentation d'un son par échantillonnage est utilisée dans de nombreux formats. Les plus simples sont RAW, WAV et BWF, mais il existe de nombreux autres formats plus sophistiqués : MP3, WMA, AAC, etc. Comme pour les images, ces formats se distinguent les uns des autres par la manière dont le son est représenté, par le fait qu'il soit compressé ou non, que le format soit public ou secret et propriétaire ou libre.

ALLER PLUS LOIN **La notation musicale et la musique contemporaine**

La comparaison des différentes méthodes de représentation des sons permet de prendre conscience de leurs spécificités. La notation musicale présente l'avantage, sur les formats RAW ou MP3, de pouvoir être lue par un musicien. En revanche, elle ne permet pas de représenter tous les sons. Il est par exemple impossible de représenter le bruit d'une locomotive ou d'une porte qui grince en duo avec un flexaton. Si les musiciens baroques ne pouvaient utiliser le bruit d'une locomotive, ou même le barrissement d'un éléphant, dans l'une de leurs compositions, les magnétophones, puis les instruments électroniques, ont permis aux compositeurs contemporains de composer des *Nocturne aux chemins de fer* et des *Variations pour une porte et un soupir*, qui mettent en évidence certaines limites de la notation musicale.

La taille d'un texte, d'une image ou d'un son

La taille d'une suite de 0 et de 1, que cette suite représente un texte, une image ou un son, s'exprime soit en *bits*, soit en *octets*, un octet étant égal à 8 bits. Par exemple, la suite 0111 0001 0010 0111 a une taille de 16 bits, ou encore de 2 octets.

Comme les textes, les images et les sons sont souvent de longues suites ; on peut exprimer leurs tailles en kilooctets, mégaoctets, gigaoctets ou téraoctets. Comme en physique, un kilo (k) est un millier (10^3), un méga (M) un million (10^6), un giga (G) un milliard (10^9) et un téra (T) mille milliards (10^{12}). Ainsi, une image d'un mégaoctet est formée de huit millions de bits. On utilise cependant souvent des préfixes similaires qui expriment des nombres ronds, non en décimal, mais en binaire :

- un kilo (binaire) est 1 024 (2^{10}),
- un méga (binaire) 1 048 576 (2^{20}),
- un giga (binaire) 1 073 741 824 (2^{30}),
- un téra (binaire) 1 099 511 627 776 (2^{40}).

ALLER PLUS LOIN **Préfixes**

Pour distinguer ces préfixes des préfixes décimaux, certains ont proposé d'utiliser plutôt les préfixes kibi (Ki), mébi (Mi), gibi (Gi) et tébi (Ti). Ainsi un mégaoctet (Mo) serait 1 000 000 octets et un mébioctet (Mio) 1 048 576 octets. Ces préfixes sont malheureusement peu souvent utilisés.

Savoir-faire Comprendre les tailles des données et les ordres de grandeurs

Trois quantités peuvent entrer en jeu dans la taille d'un fichier de données :

1. le nombre de bits utilisé pour représenter une « unité » de donnée : pixel, caractère alphanumérique, échantillon sonore, etc. ;
2. éventuellement, selon le format, la fréquence d'échantillonnage : nombre d'échantillons par seconde, de pixels par centimètre, etc. ;
3. la taille de l'objet dans une unité concrète : durée du son en secondes, surface de l'image en centimètres carrés, etc.

Exercice 9.6 (avec corrigé)

On enregistre un son pendant 10 min avec 10 000 échantillons par seconde et 16 bits pour chaque échantillon, sans compresser les données. Quelle est la taille du fichier ?

La taille du fichier est 10 × 60 × 10 000 × 16 bits = 96 000 000 bits, soit 91,55 mégabits (binaire) environ.

Exercice 9.7

On enregistre une image 10 cm × 10 cm, avec 100 pixels par centimètre, chaque pixel étant représenté par trois nombres entiers, chacun codé sur un octet. Quelle est la taille du fichier ?

Exercice 9.8

Exprimer les capacités des exercices précédents en mégabits décimaux. Que constate-t-on ? Ces presque 5 % de marge entre les capacités exprimées en décimal et en binaire permettent à quelques constructeurs de gonfler un peu les capacités des disques ou des mémoires qu'ils vendent. Voilà une raison de plus d'être vigilant quand on lit les données techniques des produits que l'on achète.



Exercice 9.9

Prendre une photo avec un petit appareil numérique, comme celui intégré à un téléphone. Observer la taille du fichier obtenu. Calculer la taille de l'image après s'être renseigné sur le nombre de pixels de l'appareil et en supposant qu'un pixel est exprimé sur trois octets. Comparer à la taille du fichier. En déduire le taux de compression.

Recommencer avec dix photos en prenant soin de prendre à la fois des vues très riches, avec beaucoup de petits détails, et des vues sans rien : une feuille blanche ou une photo sans flash dans le noir. Comparer les taux de compression obtenus.

Savoir-faire Choisir un format approprié par rapport à un usage ou un besoin, à une qualité, à des limites

Identifier les points suivants.

1. Les données doivent-elles être stockées avec précision ou un certain taux de perte est-il acceptable ?
2. Quelle taille est acceptable pour le fichier ainsi créé ?
3. Quels logiciels devront pouvoir accéder au fichier ?

Exercice 9.10 (avec corrigé)

Pour une photographie de vacances, peut-on accepter un format avec perte ? Si on doit envoyer cette photo dans un courrier électronique, quelle est la taille maximale du fichier ? Quels logiciels utiliseront les récepteurs de cette image ?

Pour une photographie de vacances, on peut accepter une légère perte. Si le fichier doit transiter par courrier, sa taille est limitée à quelques mégaoctets chez la plupart des fournisseurs d'adresses et il faut en général rechercher une taille moindre pour que le récepteur du fichier puisse y accéder en un temps raisonnable. A priori, on ne sait pas quels logiciels seront utilisés, il faut donc employer un format lisible par le plus grand nombre de logiciels différents, par exemple un format libre.

Exercice 9.11

Pour une courbe représentant les résultats d'un TP de physique que l'on souhaite présenter dans un exposé, peut-on accepter un format avec perte ? Quelle est la taille maximale du fichier ? Quels logiciels utilise-t-on pour lire cette image ?



Exercice 9.12

Ouvrir un logiciel de traitement d'images tel que Gimp et y charger une photo. Observer dans le logiciel sa taille en mémoire en choisissant la fonction *Propriété de l'image* dans le menu *Image*. Cette taille en mémoire est le produit du nombre de pixels et de la taille des pixels. En déduire la taille, en octets, de chaque pixel. Pour une photo en couleurs, le résultat devrait être 3 octets. Toutefois, si le format de l'image n'est pas PPM, le résultat peut être différent.

Enregistrer cette image sous différents formats et à différents niveaux de compression (voir le chapitre 12). Comparer visuellement les résultats obtenus. Observer si le taux de compression proposé correspond bien au rapport des tailles des fichiers.

Ai-je bien compris ?

- Quelles sont les deux principales manières de représenter une image ?
- Comment représente-t-on un son ?
- En quelle unité se mesure typiquement la taille d'un son ou d'une image ?

Les fonctions booléennes

10

Pour un oui… ou pour un non ? Oui ou non ?

Les fonctions booléennes sont utilisées partout : dans les langages de programmation, en architecture des ordinateurs, dans certains algorithmes cryptographiques…

Elles peuvent être décrites par des tables ou de manière symbolique et nous montrons comment passer d'une représentation à une autre.

Nous nous concentrons dans ce chapitre sur les fonctions *non*, *et* et *ou* qui permettent d'exprimer toutes les autres.



Dans le Manoir de Bletchey Park, quartier général des services de renseignement britanniques, **Thomas Flowers** (1905 - 1998) a construit pendant la seconde guerre mondiale la machine *Colossus*, premier calculateur électronique à utiliser le système binaire. Si cette machine n'était pas encore un ordinateur, elle a cassé les codes secrets (voir le chapitre 12) utilisés par l'armée allemande et a été un élément essentiel dans la victoire alliée. Plusieurs milliers de personnes ont travaillé à Bletchey Park, en particulier Alan Turing.

Nous nous intéressons, dans ce chapitre, aux *fonctions booléennes*, qui associent un booléen, 0 ou 1, à un ou plusieurs booléen(s).

L'expression des fonctions booléennes

Comme les fonctions d'une variable réelle, les fonctions booléennes peuvent s'exprimer de manière symbolique : l'expression *(non(x) et y) ou (x et z)* est bâtie sur le même modèle que l'expression $(\sin(x) \times y) + (x \times z)$, sauf que les variables x, y et z y représentent des booléens et non des nombres réels.

En revanche, contrairement aux fonctions d'une variable réelle, ces fonctions ne peuvent pas s'exprimer par des courbes. Néanmoins, elles peuvent s'exprimer d'une nouvelle manière : par des tables car, à la différence des nombres réels, les booléens sont en nombre fini.

non(x)

x	*non*(*x*)
0	1
1	0

et(x,y)

x	y	*et*(*x*,*y*)
0	0	0
0	1	0
1	0	0
1	1	1

ou(x,y)

x	y	*ou*(*x*,*y*)
0	0	0
0	1	1
1	0	1
1	1	1

Notation

De même que l'on écrit $x + y$ le nombre que l'on devrait, en toute rigueur, écrire $+(x,y)$, on écrit souvent *x et y* le booléen que l'on devrait écrire *et*(*x*,*y*).

Les fonctions *non*, *et*, *ou*

Les fonctions *non*, *et* et *ou* sont définies par les tables ci-contre.

Le nom de ces fonctions vient de la convention de lire 0 comme « faux » et 1 comme « vrai ».

- Ainsi, la fonction *non* transforme faux en vrai et vrai en faux : le booléen *non*(x) est donc égal à 1 si et seulement si x n'est pas égal à 1.
- De même, le booléen *x et y* est égal à 1 si et seulement si x est égal à 1 et y est égal à 1.
- Le booléen *x ou y* est égal à 1 si et seulement si au moins x ou y est égal à 1.

On remarquera que, quand x et y sont tous les deux égaux à 1, *x ou y* est égal à 1. Ce *ou* est donc inclusif ; c'est le *ou* qui apparaît dans la phrase « Je viendrai s'il y a un bus ou un métro. » et non le *ou exclusif* qui apparaît dans la phrase « Tu dois choisir : aller à la mer ou aller à la montagne. » Pour le distinguer du précédent, ce *ou exclusif* sera noté *oux*.

L'expression des fonctions booléennes avec les fonctions *non, et, ou*

On peut exprimer de manière symbolique toutes les fonctions booléennes avec les seules fonctions *non*, *et* et *ou*.

Avant de voir comment le faire dans le cas général, on commence par exprimer cinq fonctions particulières, qui seront utilisées dans la suite.

La première est la fonction *multiplexeur, mux*. Cette fonction de $\{0,1\}^3$ dans $\{0,1\}$ est telle que si x vaut 0, alors *mux(x,y,z)* vaut y et si x vaut 1, alors *mux(x,y,z)* vaut z. Elle est définie par la table ci-contre. Elle s'exprime avec les fonctions *non*, *et* et *ou*, par l'expression symbolique *mux*(x,y,z) = (*non*(x) *et* y) *ou* (x *et* z).

mux(x,y,z)

x	y	z	*mux(x,y,z)*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Pour le montrer, on calcule ligne à ligne la table de la fonction qui à x, y et z associe (*non*(x) *et* y) *ou* (x *et* z) en ajoutant des colonnes correspondant aux calculs intermédiaires et on vérifie, ligne à ligne, que cette table est celle de la fonction *mux* :

x	y	z	non(x)	non(x) et y	x et z	(non(x) et y) ou (x et z)
0	0	0	1	0	0	0
0	0	1	1	0	0	0
0	1	0	1	1	0	1
0	1	1	1	1	0	1
1	0	0	0	0	0	0
1	0	1	0	0	1	1
1	1	0	0	0	0	0
1	1	1	0	0	1	1

Les quatre autres cas particuliers sont les quatre fonctions de $\{0,1\}$ dans $\{0,1\}$:

- la fonction constante égale à 0 (❶) exprimée, de manière symbolique, par $h(x)$ = x *et non*(x),
- la fonction constante égale à 1 (❷) exprimée par $k(x)$ = x *ou non*(x),
- l'« identité » (❸) exprimée par $i(x) = x$, sans utiliser aucune fonction,
- et la fonction *non*,

qui peuvent donc toutes les quatre être exprimées avec des fonctions *non*, *et* et *ou*.

❶

x	*h(x)*
0	0
1	0

❷

x	*k(x)*
0	1
1	1

❸

x	*i(x)*
0	0
1	1

On peut maintenant voir comment exprimer toutes les fonctions de $\{0,1\}^n$ dans $\{0,1\}$ avec les fonctions *non*, *et* et *ou*. On procède par récurrence sur n.

Dans le cas $n = 1$, la fonction à exprimer est l'une des fonctions h, k, i et *non*, qui peuvent toutes s'exprimer avec les fonctions *non*, *et* et *ou*.

On suppose maintenant que l'on sait exprimer toutes les fonctions de $\{0,1\}^n$ dans $\{0,1\}$ avec les fonctions *non*, *et* et *ou* et on se donne une fonction f de $\{0,1\}^{n+1}$ dans $\{0,1\}$. On définit les fonctions g et g' de $\{0,1\}^n$ dans $\{0,1\}$ par :

```
g(x1, …, xn) = f(x1,…, xn,0)
g'(x1,…, xn) = f(x1,…, xn,1)
```

et on remarque que la fonction f peut s'exprimer de manière symbolique avec les fonctions g, g' et *mux* :

```
f(x1,…, xn, xn+1) = mux(xn+1, g(x1,…, xn), g'(x1,…, xn))
```

car, quand x_{n+1} vaut 0, le membre de gauche et le membre de droite sont tous les deux égaux à $g(x_1,\ldots, x_n)$ et, quand x_{n+1} vaut 1, les deux membres sont égaux à $g'(x_1,\ldots, x_n)$.

Par hypothèse de récurrence, on sait exprimer les deux fonctions g et g' de manière symbolique avec les fonctions *non*, *et* et *ou*, et comme on sait aussi exprimer la fonction *mux*, on peut exprimer la fonction f en remplaçant les fonctions *mux*, g et g' par leur expression en termes de *non*, *et* et *ou*.

SAVOIR-FAIRE Trouver une expression symbolique exprimant une fonction à partir de sa table

On identifie le nombre d'arguments de la fonction `f`. On extrait de la table de la fonction `f` les tables des fonctions `g` et `g'` définies par :

```
g(x1, …, xn) = f(x1,…, xn,0)
g'(x1,…, xn) = f(x1,…, xn,1)
```

On trouve les expressions symboliques de ces deux fonctions et on construit celle de la fonction `f` à partir de ces deux expressions symboliques et de celle de la fonction multiplexeur.

Exercice 10.1 (avec corrigé)

Trouver une expression symbolique exprimant une fonction *ou exclusif* (*oux*) définie par la table ci-contre.

x	y	x oux y
0	0	0
0	1	1
1	0	1
1	1	0

Le booléen x oux y *est égal à 1 si et seulement si* x *est égal à 1 ou* y *est égal à 1, mais pas les deux.*

Soit g *la fonction définie par* g(x) = x oux *0 et* g' *la fonction définie par* g'(x) = x oux *1. La table de la fonction* g *est* ❶ *et celle de la fonction* g' *est* ❷.

On reconnaît les tables de la fonction identité et de la fonction non. *Donc* g(x) = x *et* g'(x) = non(x)*, d'où on tire l'expression de la fonction* oux *:*

- *x oux y* = mux(*y,x,*non(*x*)) = (non(*y*) *et x*) ou (*y* et non(*x*))

❶

x	g(x)
0	0
1	1

❷

x	g'(x)
0	1
1	0

Exercice 10.2

Trouver l'expression symbolique de la fonction de « si et seulement si » définie par la table ci-contre.

x	y	x ssi y
0	0	1
0	1	0
1	0	0
1	1	1

L'expression des fonctions booléennes avec les fonctions *non* et *ou*

On veut maintenant montrer qu'il est possible de se passer de la fonction *et* et que toutes les fonctions booléennes peuvent s'exprimer avec les fonctions *non* et *ou*. Pour cela, il suffit de montrer que la fonction *et* elle-même peut s'exprimer ainsi. Cette fonction s'exprime de la manière suivante :

- *x et y = non* (*non*(*x*) *ou non*(*y*))

En effet, la table de la fonction qui à *x* et *y* associe *non* (*non*(*x*) *ou non*(*y*)) se calcule ligne à ligne (voir ci-contre) et on reconnaît la table de la fonction *et*.

x	y	non (non(x) ou non(y))
0	0	0
0	1	0
1	0	0
1	1	1

Exercice 10.3

Montrer que :

- *x ou y = non* (*non*(*x*) *et non*(*y*))

En déduire que toutes les fonctions booléennes peuvent s'exprimer avec les fonctions *non* et *et*.

Exercice 10.4

x	y	S (x,y)
0	0	1
0	1	1
1	0	1
1	1	0

La fonction de Sheffer exprime l'incompatibilité de deux valeurs booléennes. Elle est définie par la table ci-contre.

Montrer que *S(x, y) = non(x et y)*.

Montrer, réciproquement, que *non*(*x*) = *S*(*x*,*x*) et *x ou y* = *S*(*S*(*x*,*x*),*S*(*y*,*y*)). En déduire que toutes les fonctions booléennes peuvent s'exprimer avec la fonction de Sheffer uniquement.

Le choix d'exprimer les fonctions avec les fonctions *non* et *ou* n'est donc qu'un choix parmi d'autres.

Exercice 10.5

Quand deux interrupteurs sont en parallèle, la lumière s'allume quand l'un d'eux est fermé. Quand ils sont en série, la lumière s'allume quand les deux sont fermés. Quand ils sont en va-et-vient la lumière s'allume quand les deux sont fermés ou les deux sont ouverts. Donner la table de la fonction booléenne dans ces trois cas et exprimer ces trois fonctions booléennes avec les fonctions *non* et *ou*.

Exercice 10.6

Montrer que *x et y* = *y et x*. Est-ce la même chose pour *ou* ? Calculer *x et 1* et *x et 0*. On appelle *élément neutre* d'une fonction binaire *f*, un élément *n* tel que pour tout *x*, *f(x,n)* = *f(n,x)* = *x* et *élément absorbant* un élément *a* tel que pour tout *x*, *f(x,a)* = *f(a,x)* = *a*. La fonction *et* a-t-elle un élément neutre ? Et un élément absorbant ? La fonction *ou* a-t-elle un élément neutre ? Et un élément absorbant ?



Exercice 10.7

Med est content si Bob et Jon sont tous les deux là, mais sans Rut, ou si Rut est là soit avec Bob, soit avec Jon. Construire une table avec en entrée la présence de Rut, Bob et Jon et en sortie le booléen qui vaut 1 si Med est content. Exprimer le choix de Med par une phrase plus simple.



Exercice 10.8

Soit *f* une fonction qui s'exprime de manière symbolique avec la fonction *ou* uniquement. Montrer que *f*(0,0,…,0) = 0. Montrer que la fonction *non* ne peut pas s'exprimer avec la fonction *ou* uniquement.

Ai-je bien compris ?

- Quelles sont les deux principales manières de représenter une fonction booléenne ?
- Quelle est la table de la fonction *non* ? Celle de la fonction *et* ? Celle de la fonction *ou* ?
- Quelles sont les fonctions de base à partir desquelles il est possible de construire toutes les autres fonctions ?

Structurer l'information

11

Chapitre avancé



En 1989, **Tim Berners-Lee** (1955 –) a proposé un outil aux nombreux chercheurs de l'Organisation européenne pour la recherche nucléaire (CERN) pour partager de grandes quantités d'informations : insérer dans des textes des liens vers d'autres textes, situés sur d'autres ordinateurs, auxquels on accède à travers le réseau Internet. Cette toile d'araignée de liens a vite trouvé un nom : le Web. Tim Berners Lee est l'auteur du langage HTML et du premier navigateur : Nexus.

Comment trouver son chemin dans une jungle d'informations ?

Dans ce chapitre, nous voyons comment les informations s'effacent, se conservent, s'organisent, selon les usages que nous voulons en faire, de la notion de fichier en arborescence et de liens à celle de base de données.

Nous insistons ici sur la notion de persistance des données avec le danger de l'hypermnésie et discutons les enjeux autour de la « gratuité » de la copie ou diffusion de l'information.

La persistance des données

Reprenons l'idée de créer, comme au chapitre 3, un programme de gestion d'un répertoire. Cependant, au lieu d'être décidés une fois pour toutes au moment de l'écriture du programme, les noms et les numéros de téléphone de ce répertoire sont entrés par l'utilisateur du programme.

Par exemple :

```
a Alice 060606060606
Contact ajouté
```
On ajoute un numéro.

```
a Bob 0606060607
Contact ajouté
```
On ajoute un autre numéro.

```
i Alice
60606060606
```
On interroge le répertoire.

```
q
```
On quitte le programme.

Le point qui nous intéresse ici est que, quand on tape la commande `q`, le programme se termine et les données entrées sont perdues. Si on exécute à nouveau ce programme, on doit à nouveau entrer tous les contacts, ce qui n'est pas en général ce que l'on souhaite faire quand on utilise un répertoire : on souhaite que les données soient *persistantes*, c'est-à-dire qu'elles demeurent à un endroit accessible quand le programme se termine et même quand on éteint l'ordinateur sur lequel ce programme est exécuté, voire quand on remplace cet ordinateur par un autre.

Cette question de la persistance des données se pose à deux niveaux. C'est d'abord une question de matériel : les valeurs stockées dans les variables d'un programme sont physiquement stockées dans la mémoire de l'ordinateur (voir le chapitre 14) et ces données sont perdues quand l'ordinateur est éteint et que la mémoire cesse d'être alimentée en courant électrique. Cela a mené à concevoir des périphériques, comme les *disques*, ou les *clés de mémoire flash* aussi appelées *clés USB*, qui peuvent stocker des données de manière persistante. La persistance des données est ensuite une question de programmation, puisqu'il faut être capable, dans un programme, de stocker des données sur un tel périphérique et d'utiliser de telles données.

La notion de fichier

Un texte en ASCII ou en HTML, une image au format PBM, PGM ou PPM, un son au format RAW ou MP3, un programme en Java ou en C, sont des exemples de données que l'on souhaite faire persister, par exemple en les stockant sur un disque.

Un disque stocke simplement une suite de bits, une suite de 0 et de 1. Le nombre de bits qu'un disque peut stocker est appelé sa *capacité* : par exemple un disque d'un téraoctet (binaire) peut stocker 2^{40} mots de 8 bits, soit un peu plus de huit mille milliards de bits. On peut donc facilement stocker un texte, une image, un son ou un programme sur un tel disque. Cependant, comme on souhaite souvent stocker sur un disque plusieurs images, textes, etc., il faut diviser les huit mille milliards de bits dont le disque est constitué en plusieurs espaces plus petits, que l'on appelle des *fichiers*. Un fichier est simplement une suite de 0 et de 1, à laquelle on associe un nom. Par exemple, nous avons vu que le texte « Je pense, donc je suis. » se représente en ASCII comme la suite de 184 bits suivante :

```
010010100110010100100000011100000110010101101110011100110110010
100101100001000000110010001101111011011100110001100100000011010
1001100101001000000111001101110101011010010111001100101110
```

Il est possible de stocker cette suite de bits sur un disque en lui donnant le nom `cogito.txt`, l'extension `txt` indiquant que cette suite de bits exprime un texte en ASCII.

Utiliser un fichier dans un programme

Pour revenir à l'exemple du répertoire, on commence par stocker les contacts dans un fichier `repertoire.txt` qui peut avoir été écrit à la main, en utilisant un éditeur de texte ou un logiciel de traitement de texte, ou été produit par un autre programme. Il a la forme suivante :

```
Alice
0606060606
Bob
0606060607
```

```
Charles
0606060608
Djamel
0606060609
Étienne
0606060610
Frédérique
0606060611
Guillaume
0606060612
Hector
0606060613
Isabelle
0606060614
Jérôme
0606060615
```

Le programme d'interrogation du répertoire est semblable à celui du chapitre 3, à la différence près que, au lieu de remplir les cases des tableaux `nom` et `tel` avec des constantes, on transfère ces informations depuis le fichier `repertoire.txt` dans les tableaux `nom` et `tel`. Pour cela, on utilise une nouvelle fonction `Isn.readStringFromFile` (lire une chaîne de caractères dans un fichier) en tout point semblable à la fonction `Isn.readString` que l'on a déjà employée, sauf qu'au lieu d'attendre qu'une chaîne de caractères soit tapée au clavier, elle la lit dans un fichier. Avant de pouvoir lire dans un fichier, il faut établir un canal de communication avec lui ; c'est ce que l'on appelle *ouvrir* le fichier. Cela se fait avec la fonction `Isn.openIn` qui prend en argument une chaîne de caractères, le nom du fichier, et qui retourne le canal de communication lui-même. Ensuite, la fonction `Isn.readStringFromFile` prend en argument ce canal de communication et retourne la chaîne de caractères lue. Quand la lecture est achevée, il faut fermer le canal de communication avec l'instruction `Isn.closeIn`. Le canal de communication lui-même a le type `java.util.Scanner`.

En résumé, le programme commence par allouer deux tableaux `nom` et `tel`, ouvre un canal de communication `f` avec le fichier `repertoire.txt`, lit les dix noms et les dix numéros de téléphone sur ce canal de communication, puis ferme ce canal.

```
nom = new String [10];
tel = new String [10];
f = Isn.openIn("repertoire.txt");
for (i = 0; i <= 9; i = i + 1) {
  nom[i] = Isn.readStringFromFile(f);
  tel[i] = Isn.readStringFromFile(f);}
Isn.closeIn(f);
```

La suite du programme est similaire à celui du chapitre 3. On demande un nom à l'utilisateur, au clavier cette fois-ci, on recherche ce nom dans le tableau `nom` à l'aide d'une boucle `while` et quand on l'a trouvé, on affiche le numéro de téléphone correspondant.

```
s = Isn.readString();
i = 0;
while (i < 10 && !Isn.stringEqual(s,nom[i])) {
  i = i + 1;}
if (i < 10) {
  System.out.println(tel[i]);}
else {
  System.out.println("Inconnu");}
```

On peut alors utiliser ce programme comme celui du chapitre 3 :

```
Hector
0606060613
```

Exercice 11.1

Que se passe-t-il si l'on ouvre et ferme le fichier à chaque lecture ?

Exercice 11.2

Écrire un programme de répertoire inversé, qui demande à l'utilisateur un numéro de téléphone et recherche le nom associé.

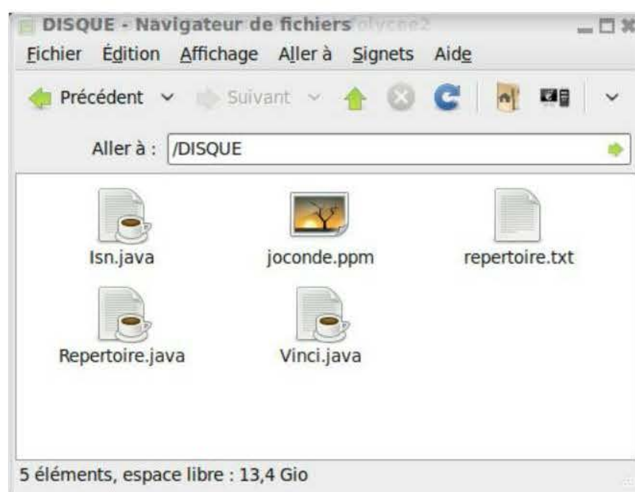
Écrire dans un fichier se fait de manière similaire : on ouvre le fichier avec l'instruction `Isn.openOut` qui prend en argument une chaîne de caractères et qui retourne un canal de communication. On écrit dans le fichier avec les instructions `Isn.printToFile` et `Isn.printlnToFile` qui prennent en argument un canal de communication et un objet à écrire, chaîne de caractères, nombre entier ou nombre à virgule, et on ferme le canal de communication avec l'instruction `Isn.closeOut`. Un canal de communication qui permet, non de lire, mais d'écrire dans un fichier a le type `java.io.OutputStreamWriter`.

Exercice 11.3

Adapter le programme, ci-avant, pour qu'il puisse enregistrer de nouveaux numéros. Le fichier sera entièrement lu et réécrit à chaque modification.

Organiser des fichiers en une arborescence

Quand un disque, ou une clé de mémoire flash, contient plusieurs fichiers, il est possible d'en afficher la liste. La manière la plus courante de le faire est de représenter chaque fichier par une icône dans une fenêtre, la forme de l'icône variant en fonction du format du fichier.

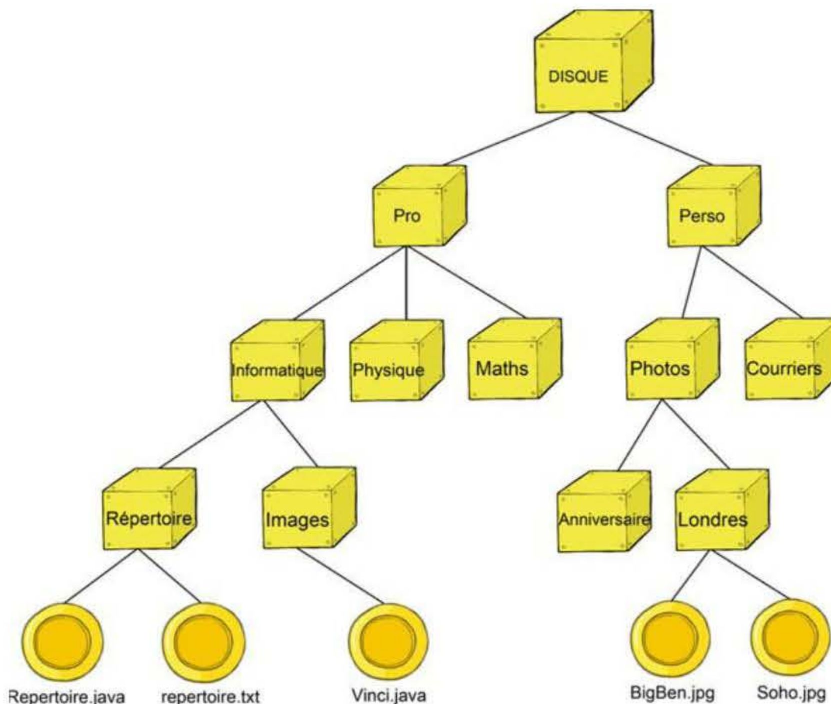


On peut aussi simplement afficher la liste des fichiers par ordre alphabétique avec, dans une fenêtre *terminal*, une commande qui s'appelle `ls` ou `dir` selon le système d'exploitation.

```
ls
Isn.java joconde.ppm Repertoire.java repertoire.txt Vinci.java
```

Découper un disque en fichiers n'est toutefois pas suffisant, car il est probable que ce disque contiendra rapidement plusieurs milliers de fichiers : des fichiers professionnels, comme les programmes que l'on est en train ou que l'on a terminé de développer, des fichiers personnels, comme des photos de vacances, etc. Il est donc nécessaire d'organiser ces fichiers. Une manière de le faire est de regrouper ces derniers dans des *dossiers* ; par exemple, tous les fichiers professionnels dans un dossier *Pro* et tous les fichiers personnels dans un dossier *Perso*. À l'intérieur du dossier *Pro*, on peut encore regrouper tous les fichiers qui concernent ce cours dans un dossier *Informatique*, tous les fichiers qui concernent le cours de physique dans un dossier *Physique*, etc. À l'intérieur du dossier

Informatique, on peut regrouper tous les fichiers qui concernent le projet d'écrire un programme de gestion de répertoire dans un dossier *Répertoire*, etc. Si bien que le disque contient un dossier *Pro*, qui contient un dossier *Informatique*, qui contient un dossier *Répertoire*, qui contient enfin le fichier `Repertoire.java`, et d'autres. On appelle une telle organisation des fichiers une organisation *arborescente*, car on peut la visualiser sous la forme d'un arbre.



Dans cet arbre, le *chemin* d'un fichier est la liste des noms des nœuds de la branche qui va de la racine de l'arbre au fichier en question. Par exemple, le chemin du fichier `Repertoire.java` est `/DISQUE/Pro/Informatique/Répertoire/Repertoire.java`.

SAVOIR-FAIRE **Classer des fichiers sous la forme d'une arborescence**

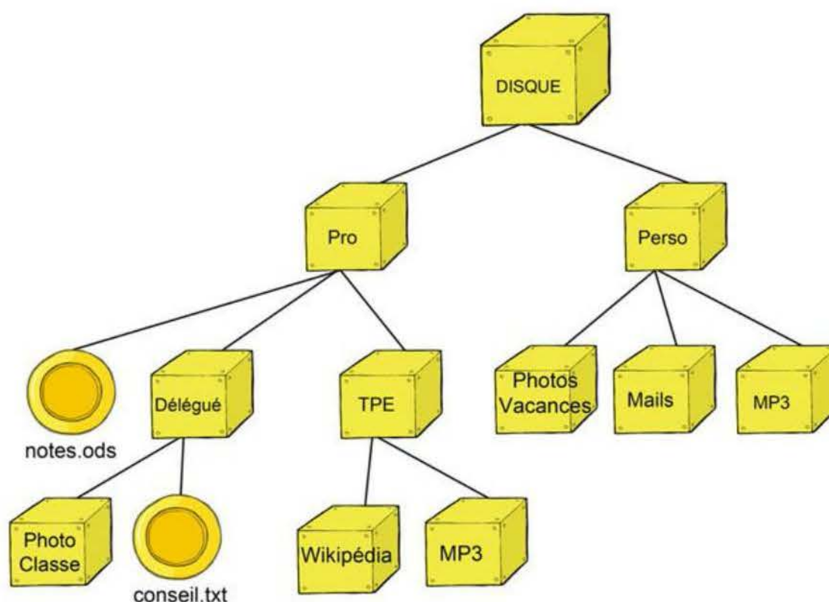
Faire la liste des fichiers à classer, les regrouper en catégories homogènes, donner un nom significatif à chacune de ces catégories. Éventuellement les regrouper elles-mêmes en catégories homogènes. Créer les dossiers correspondant à chacune de ces catégories. Mettre chaque fichier dans le dossier approprié.

Exercice 11.4 (avec corrigé)

Classer, sous la forme d'une arborescence, les fichiers suivants :

- des photos de vacances,
- des photos de sa classe,
- une page de tableur présentant ses notes du trimestre,
- des pages de Wikipédia présentant des informations utiles pour son TPE,
- des copies de mails personnels,
- un fichier texte contenant les notes prises dans une réunion préparatoire au conseil de classe,
- des fichiers de musique pour son baladeur,
- des fichiers de musique téléchargés en préparation de son TPE.

On commence par donner un nom à chacune de ces catégories : photos de vacances, photos de classe, etc. On peut regrouper les dossiers contenant les photos de vacances, les mails personnels et les fichiers pour son baladeur, dans un dossier Perso, les photos de la classe et les notes prises en réunion dans un dossier Délégué, puisque ces différentes informations sont liées à cette fonction, les pages de Wikipédia et la musique téléchargée pour le TPE peuvent être rassemblées dans un dossier TPE, enfin on peut regrouper le dossier Délégué, le dossier TPE et les notes du trimestre dans un dossier Pro. On obtient alors l'arborescence suivante.



Exercice 11.5

Classer, sous la forme d'une arborescence, les fichiers qui traînent dans son répertoire personnel sur les ordinateurs du lycée.

Liens et hypertextes

L'organisation arborescente des fichiers n'est pas le seul moyen de structurer l'information : elle est en concurrence avec d'autres méthodes, parmi lesquelles l'utilisation de liens *hypertextes*, notion qui n'a pas été inventée pour structurer l'information, mais pour simplifier la notion de référence dans une page web (voir le chapitre 8).

Ainsi, une page web, écrite au format HTML,

```
Le programme <a href =
"file:///DISQUE/Pro/Informatique/Répertoire/
Repertoire.java">Repertoire.java</a> permet de rechercher un
nom dans un répertoire, exprimé sous forme d'un fichier texte,
comme le fichier
<a href =
"file:///DISQUE/Pro/Informatique/Répertoire/
repertoire.txt">repertoire.txt</a>.
```

visualisée dans un navigateur, apparaît de la manière suivante :

> Le programme Repertoire.java permet de rechercher un nom dans un répertoire, exprimé sous forme d'un fichier texte, comme le fichier repertoire.txt.

Et en cliquant sur l'un des mots en bleu et soulignés, on accède directement au fichier `Repertoire.java` ou au fichier `repertoire.txt`. Il est donc possible d'accéder à un fichier sans savoir précisément où il se trouve dans l'arborescence, simplement en cliquant sur un lien.

Cette remarque mène à une autre manière d'organiser les fichiers sur un disque, où la place du fichier dans l'arborescence est moins importante que la manière d'y accéder en cliquant sur un lien qui apparaît dans une page. Par exemple, au lieu de classer ses photos dans plusieurs dossiers, *Anniversaire*, *Londres*, etc. on laisse ses photos en vrac dans un dossier et on crée une page web pour accéder à ses photos d'anniversaire, une autre pour accéder aux photos de son voyage à Londres, etc. et une page web qui permet d'accéder à chacune de ces pages. Cette idée est à la base du Web, des logiciels de gestion de photos ou de fichiers son et des réseaux sociaux.

EN SAVOIR PLUS Structure d'arbre et structure de graphe

Cette méthode permet aussi de dépasser facilement les limites d'un disque unique et de référencer des fichiers qui se trouvent ailleurs sur le réseau. En outre, dans une structure arborescente, si le dossier *B* est un élément du dossier *A*, le dossier *A* ne peut pas à son tour être un élément du dossier *B*. En revanche, avec des liens hypertextes, rien n'empêche une page A de contenir un lien vers une page B, qui contient elle-même un lien vers la page A. À une structure d'arbre se substitue donc une structure de *graphe* (voir le chapitre 22).

L'hypermnésie

La persistance des données, qui est souvent souhaitée, a aussi quelques effets indésirables. Par exemple, quand un ordinateur hors d'usage est mis au grenier, son disque continue à contenir des milliers de textes, courriers, photos, etc. et il est possible que cent ans plus tard, ces informations soient encore accessibles. Les ordinateurs sont *hypermnésiques* : ils n'oublient rien. Souhaite-t-on réellement que ses petits enfants puissent accéder à toutes ses photos, à la liste des sites web que l'on a visités, aux conversations que l'on a eues sur le chat, aux mails que l'on a échangés, etc. ?

Bien entendu, ce phénomène d'hypermnésie existait avant les ordinateurs : dans le même grenier, un album de photos ou une liasse de lettres manuscrites pouvaient aussi voyager dans le temps. La différence est qu'un album de photos ou une liasse de lettres se voit et contient peu d'informations. Un disque de un téraoctet (binaire) peut stocker 2^{40} caractères, soit un million de livres de cinq cents pages, sans que l'on voie, au premier abord, ce qu'il contient.

Si l'on n'y prend garde, on ira peut-être vers un monde dans lequel rien ne s'oubliera : chaque geste laissera une trace dans tout état postérieur du monde.

Cette hypermnésie des ordinateurs serait un problème facile à résoudre si les ordinateurs n'étaient pas connectés en réseau : chacun devrait simplement trier les informations qu'il souhaite garder et celles qu'il souhaite détruire sur le disque de son ordinateur. Néanmoins, comme les ordinateurs sont connectés en réseaux, les courriers échangés peuvent aussi être conservés par les fournisseurs de services de courrier. Il en va de même avec la liste des produits que l'on achète et les magasins en ligne, ou ceux dans lesquels on a une carte de fidélité, les images filmées par les caméras de surveillance et les entreprises qui installent et gèrent ces caméras, etc.

Ces problèmes, qui sont nouveaux, ont déjà reçu des solutions partielles, mais bien des solutions sont encore à inventer. Parmi elles, certaines sont individuelles : quand on met une photo en ligne, on peut la protéger par un mot de passe ou en restreindre l'accès à un petit nombre de personnes ; ainsi elle ne sera pas archivée par les moteurs de recherche. D'autres sont collectives : par exemple, l'article 6 de la loi du 6 janvier 1978 (modifiée plusieurs fois) indique que des données à caractère personnel ne peuvent être conservées sous une forme permettant l'identification des personnes concernées au-delà de la durée nécessaire aux finalités pour lesquelles elles ont été collectées.

Sujet d'exposé — Loi du 6 janvier 1978

Rechercher le texte de la loi du 6 janvier 1978 afin d'en présenter les idées principales.

Alors que, dans les derniers millénaires, l'humanité a beaucoup cherché à laisser des traces de ses actions, elle commence juste à prendre conscience de l'importance qu'il y a aussi à parfois effacer certaines de ces traces.

Exercice 11.6

Chercher sur le Web la définition du « droit à l'oubli » que certains cherchent à promouvoir comme un droit fondamental.

Sujet d'exposé La CNIL

Présenter la Commission nationale de l'informatique et des libertés.

Pourquoi l'information est-elle souvent gratuite ?

Si l'on achète une pomme et si on la mange, cette pomme ne peut pas aussi être mangée par quelqu'un d'autre. De même, si on achète un sac de charbon pour chauffer sa maison, ce sac de charbon ne peut pas aussi chauffer la maison de quelqu'un d'autre. Et si l'on achète les services d'un jardinier pour tondre sa pelouse, ce jardinier ne peut pas en même temps tondre la pelouse de quelqu'un d'autre. Une pomme, un sac de charbon ou les services d'un jardinier sont des biens dont la consommation par une personne exclut la consommation par une autre. On dit que de tels biens sont *rivaux*. Jusqu'au XXe siècle, presque tous les biens produits par l'agriculture et l'industrie étaient rivaux et notre économie, en particulier notre notion de propriété, est construite pour la production, l'échange et la consommation de tels biens.

Un fichier qui contient un texte, un morceau de musique, une image, une vidéo ou un programme est, en revanche, un bien non rival. Recopier un tel fichier ou le diffuser sur un réseau ne coûte pratiquement rien, si bien que le fait qu'une personne écoute un morceau de musique n'empêche nullement une autre personne d'écouter le même morceau. L'information est, par nature, un bien non rival. Un livre, un CD, une photo, un DVD, un CD-ROM sont des biens rivaux, mais non un texte, une pièce de musique, une image, une vidéo ou un programme.

Si une mine produit un sac de charbon et le vend à une personne qui l'utilise pour chauffer sa maison et qu'une seconde personne souhaite aussi acheter un sac de charbon pour chauffer la sienne, la mine doit produire un second sac de charbon, ce qui lui coûte de l'argent. De ce fait, la mine n'a pas d'autre choix que celui de vendre ce sac de charbon à la seconde personne. En revanche, si une entreprise produit une vidéo pour une personne et qu'une seconde personne souhaite aussi regarder cette vidéo,

l'entreprise peut, en théorie, donner cette vidéo gratuitement à la seconde personne, car, une vidéo étant un bien non rival, cela ne coûte rien de plus à l'entreprise de laisser une seconde personne en profiter également.

Ce raisonnement mène cependant à un paradoxe : pourquoi le premier client paierait-il la vidéo, s'il sait qu'il lui suffit d'attendre que quelqu'un d'autre la paie pour être le second ? Au cours de l'histoire, les producteurs de biens non rivaux ont imaginé différentes manières de répondre à cette question.

- Imiter le marché des biens rivaux, c'est-à-dire empêcher les personnes qui ne le paient pas de consommer un bien, même si cela ne coûterait rien de plus de les laisser le consommer. Ainsi l'accès à certaines chaînes de télévision est interdit aux personnes qui ne paient pas un abonnement, même si laisser ces personnes regarder ces chaînes ne coûterait rien de plus.
- Faire payer à chacun ce qu'il est prêt à payer pour ce bien. C'est par exemple le cas de logiciels qui sont payants pour les entreprises, mais beaucoup moins chers pour les particuliers, voire gratuits pour les étudiants.
- Faire payer à chacun ce qu'il souhaite payer. C'est le cas de certaines radios dans le monde, qui sont gratuites mais rappellent fréquemment à l'antenne que, si on ne leur envoie pas un chèque de temps en temps, elles finiront par fermer.
- Échanger ce bien, non contre de l'argent, mais contre un bien dont tout le monde dispose : du temps et de l'attention. C'est la réponse des chaînes de télévision ou des sites web qui sont gratuits, mais qui « demandent » à leurs consommateurs d'accorder un peu de temps et d'attention à des publicités.
- Distribuer un bien gratuitement, car cela crée de la demande pour un autre bien, rival cette fois-ci. Ainsi, certaines entreprises distribuent un logiciel gratuitement, mais font payer les cours pour apprendre à s'en servir.
- Faire payer tout le monde, consommateur ou non, c'est par exemple le cas des chaînes de télévision publiques, qui sont payées par un impôt spécial. C'est également le cas du service des pompiers qui est payé par les impôts.
- Une dernière réponse est celles d'entreprises qui développent un logiciel pour leur besoins propres et qui ensuite laissent tout le monde en profiter et aussi… améliorer ce logiciel, ce dont elles bénéficient en retour.

Plusieurs de ces réponses mènent donc à distribuer gratuitement des biens non rivaux, en particulier de l'information, ce qu'il est impossible de faire avec des pommes, des sacs de charbons ou les services d'un jardinier qui, avant d'avoir un prix, ont un coût.

Aller plus loin Les bases de données

Pour gérer de grandes quantités de données, par exemple l'ensemble des réservations de billets de train ou d'avion que doit gérer une compagnie ferroviaire ou aérienne, il n'est pas possible d'utiliser une simple structure arborescente ou à base de liens hypertextes. On doit utiliser des outils plus sophistiqués : un *système de gestion de bases de données* et un *langage de manipulation de données*, comme le langage SQL.

Le répertoire que nous avons construit ci-avant peut être défini comme un ensemble fini de couples formés d'un nom et d'un numéro de téléphone

```
R = {(nom = Alice ; tel = 0606060606),
(nom = Bob ; tel = 0606060607),
(nom = Charles ; tel = 0606060608),
(nom = Djamel ; tel = 0606060609),
(nom = Étienne ; tel = 0606060610),
(nom = Frédérique ; tel = 0606060611),
(nom = Guillaume ; tel = 0606060612),
(nom = Hector ; tel = 0606060613),
(nom = Isabelle ; tel = 0606060614),
(nom = Jérôme ; tel = 0606060615)}
```

Un tel ensemble de couples, de triplets ou, plus généralement, de n-uplets, s'appelle une *relation* et un ensemble de relations s'appelle une *base de données*.

Par exemple, si Alice, Bob, Charles, Djamel, Étienne, Frédérique, Guillaume, Hector, Isabelle et Jérôme sont musiciens, outre la relation ci-avant qui indique le numéro de téléphone de chacun, on peut définir une autre relation qui indique les instruments dont chacun joue.

```
I = {(nom = Alice ; instrument = alto),
(nom = Bob ; instrument = contrebasse),
(nom = Charles ; instrument = cor),
(nom = Charles ; instrument = trompette),
(nom = Djamel ; instrument = piano),
(nom = Étienne ; instrument = xylophone),
(nom = Frédérique ; instrument = harpe),
(nom = Guillaume ; instrument = basson),
(nom = Hector ; instrument = basson),
(nom = Hector ; instrument = contrebasson),
(nom = Isabelle ; instrument = hautbois),
(nom = Isabelle ; instrument = clarinette),
(nom = Jérôme ; instrument = violon)}
```

et une autre qui indique la famille de chaque instrument

```
F = {(instrument = violon ; famille = cordes),
(instrument = hautbois ; famille = bois),
(instrument = clarinette ; famille = bois),
(instrument = xylophone ; famille = percussions), …}
```

L'ensemble de ces trois relations constitue une base de données.

Un *système de gestion de bases de données* est un système qui permet de créer de telles relations, d'ajouter ou de retirer des n-uplets dans une relation et de rechercher des n-uplets. Pour cela, on formule des *requêtes* dans un *langage de gestion de données*.

Par exemple, on peut formuler la requête de chercher les instruments dont joue Hector, en cherchant les couples de la relation `I` dont la composante *nom* est *Hector* :

```
(nom = Hector ; instrument = basson)
(nom = Hector ; instrument = contrebasson)
```

On peut de même chercher les joueurs de basson, en cherchant les couples de la relation `I` dont la composante *instrument* est *basson*. On trouvera alors deux couples

```
(nom = Guillaume ; instrument = basson)
(nom = Hector ; instrument = basson)
```

On peut aussi fabriquer, à partir des relations `R` et `I`, une nouvelle relation qui est un ensemble de triplets formés d'un nom, d'un numéro de téléphone et d'un instrument. On appelle cela la *jointure* des relations `R` et `I` et il est possible de chercher dedans comme dans les relations simples ; par exemple, on peut chercher ceux des triplets de cette jointure dont la composante *instrument* est *basson*. On obtiendra ainsi deux triplets

```
(nom = Guillaume ; tel = 0606060612 ; instrument = basson)
(nom = Hector ; tel = 0606060613 ; instrument = basson)
```

ce qui est l'information dont on a besoin quand on cherche à remplacer un bassoniste.

Quand on utilise un système de gestion de bases de données, on exprime donc, dans un langage de haut niveau, des requêtes dont l'exécution consulte et modifie des fichiers. Toutefois, il n'est plus nécessaire de connaître la forme exacte de ces fichiers, car on n'y accède plus que par l'intermédiaire du système de gestion de bases de données.

ALLER PLUS LOIN Transformer, stocker et transmettre des informations

L'algorithme de l'addition en base deux (voir le chapitre 18) illustre une manière d'utiliser les ordinateurs pour transformer des informations : les informations 10 et 11 sont transformées en 101. Ce chapitre illustre quant à lui une autre manière d'utiliser des ordinateurs, non pour transformer des informations, mais pour les stocker et les retrouver plus tard. Une troisième utilisation des ordinateurs consiste à transmettre des informations d'un endroit à un autre.
Les ordinateurs ont essentiellement été inventés pour transformer des informations. Le stockage et la transmission sont venus plus tard et ont apporté avec eux de nouveaux problèmes et de nouveaux algorithmes, par exemple pour interroger les bases de données ou pour mettre à jour des tables de routage (voir le chapitre 16).

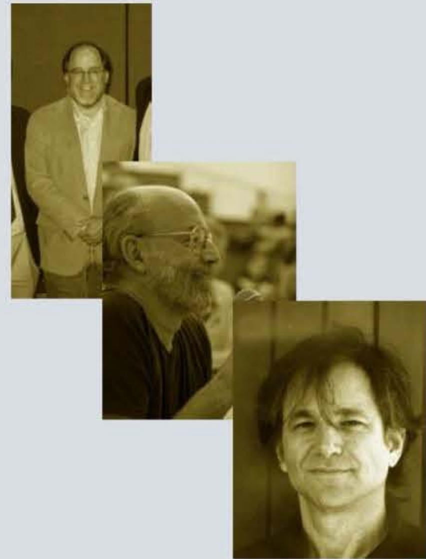
Ai-je bien compris ?

- Quelle est la différence entre une donnée persistante et une donnée non persistante ?
- Quels sont les différents moyens d'organiser les informations ?
- Que signifie le mot « hypermnésie » ?

Compresser, corriger, chiffrer

12

CHAPITRE AVANCÉ



Ronald Rivest (1947-), **Adi Shamir** (1952-) et **Len Adleman** (1945-) ont conçu, en 1978, une méthode de chiffrement, la méthode RSA, fondée sur l'utilisation de deux clés : une clé privée et une clé publique. La méthode la plus rapide connue à ce jour pour retrouver la clé privée à partir de la clé publique est la décomposition d'un nombre entier en un produit de facteurs premiers, calcul qui demande un temps très long, quand le nombre à factoriser dépasse quelques milliers de chiffres binaires.

« Zhqm, zmgm, zmfm. » (Jules César, 47 av. J.-C.)

Nous voyons maintenant comment modifier l'expression des données dans le but d'économiser de l'espace, de corriger des erreurs ou de les protéger. Pour cela, nous utilisons des formats beaucoup plus sophistiqués que ceux vus aux chapitres 8 et 9.

Pour la compression, nous expliquons comment définir un dictionnaire des mots utilisés et les coder selon leur fréquence. Pour détecter et corriger les erreurs, nous expliquons comment utiliser la redondance d'informations et exploiter des bits de contrôle de cohérence. Pour le chiffrement, nous expliquons les méthodes à clé et introduisons la notion de *clé publique – clé privée*.

Pour représenter des informations, par exemple un texte, sous la forme d'une suite de 0 et de 1, nous avons présenté les formats standards, comme ASCII ou UTF-8 (voir le chapitre 8). Il est également possible d'utiliser un format particulier pour rendre la suite de bits exprimant ces informations moins volumineuse, ou intelligible même en cas d'erreurs de transmissions, ou au contraire inintelligible, sauf pour son destinataire. Un tel format s'appelle un *code*.

Compresser

Compresser des informations

On appelle *compresser* des informations le fait de les exprimer sous la forme d'une suite de bits, avec un code choisi pour rendre cette suite aussi courte que possible.

La phrase « je pense, donc je suis. » peut se représenter en ASCII par la suite 106, 101, 32, 112, 101, 110, 115, 101, 44, 32, 100, 111, 110, 99, 32, 106, 101, 32, 115, 117, 105, 115, 46 (voir le chapitre 8). Chacun de ces nombres est exprimé sur 8 bits. Il faut donc 23 × 8 = 184 bits pour représenter cette phrase en entier.

Toutefois, seuls 13 symboles sont utilisés dans cette phrase : le « c », le « d », le « e », le « i », le « j », le « n », le « o », le « p », le « s », le « u », l'espace, la virgule et le point. On peut donc modifier le code, en représentant chaque symbole sur 4 bits et il ne faut alors plus que 23 × 4 = 92 bits pour représenter cette phrase.

On peut faire encore mieux. Le mot « je » suivi d'un espace est répété deux fois dans la phrase ; on peut donc représenter chaque symbole sur 4 bits, en convenant, de plus, que le mot entier « je » suivi d'un espace est représenté également par une suite de 4 bits. Il suffit désormais de 19 × 4 = 76 bits.

ALLER PLUS LOIN
Joindre le dictionnaire au message

Un dictionnaire étant construit sur mesure pour chaque message, il faut l'adjoindre au message pour rendre la lecture de ce dernier possible. Ce coût est cependant vite amorti quand le message est long.

Cette idée est à la base des méthodes de *compression par dictionnaire*. Un dictionnaire est une fonction qui associe des suites de bits non pas à des symboles isolés, mais à des suites de symboles. La suite de bits associée par un dictionnaire à une suite de symboles est appelée sa *référence*. Remplacer, dans un texte, chaque suite de symboles par sa référence exprime donc ce texte en binaire, et remplacer chaque référence par la suite à laquelle elle est associée permet de retrouver le texte original. En associant une référence aux suites longues et répétées dans le texte original, on obtient une suite de bits plus courte que si on avait exprimé le texte en binaire, caractère par caractère.

On peut encore améliorer le résultat en tirant parti des différences de fréquence d'apparition des symboles et suites de symboles. En effet, dans la phrase « je pense, donc je suis. », le « s » est utilisé 3 fois, le « n », l'espace, le

« e » et la séquence de symboles « je » sont utilisés 2 fois chacun, tandis que le « c », le « d », le « i », le « o », le « p », le « u », la virgule et le point sont utilisés 1 fois seulement. On peut donc réduire encore la taille du message en utilisant des références plus courtes pour les suites les plus fréquentes et plus longues pour les suites les plus rares, par exemple :

« s » : 000

« c » : 00100

« d » : 00101

« i » : 00110

« o » : 00111

« e » : 010

« je » : 011

« p » : 1000

« u » : 1001

« , » : 1010

« . » : 1011

« n » : 110

«   » : 111

Ainsi, la phrase « je pense, donc je suis. » se représente par la suite de 69 bits suivante : 0111000010110000010101011100101001111100010 01110110001001001100001011. Un code qui utilise ainsi des références de longueurs différentes est appelé un *code de Huffman*.

Quand une suite de bits représente un texte en ASCII, la découper en symboles est facile, puisque chaque symbole est représenté sur huit bits. En revanche, quand les symboles sont représentés par des références de longueurs différentes, la tâche est plus ardue. Par exemple, en Morse, où la lettre « e » se code par « . » et la lettre « i » par « .. », la suite « .. » peut ou bien représenter le message « i » ou bien le message « ee ». On dit que le code de la lettre « e » est un *préfixe* de celui de la lettre « i », car c'est le début du code de la lettre « i ». Le code Morse est donc ambigu et il est nécessaire d'utiliser un séparateur pour lever cette ambiguïté : « .. » pour « i », et « . / . » pour « ee », ce qui a l'inconvénient d'allonger la représentation du texte, alors qu'on cherche précisément à la raccourcir.

Cependant, avec le code ci-avant, on peut se passer de séparateurs, car aucun symbole n'a pour référence un préfixe de la référence d'un autre symbole. Ainsi, quand on décode la suite 0111000010110000 01010101110010100111110001001110110001001001100001011 par exemple, aucun symbole n'ayant pour référence 0, 01, 0111 ou 01110, le

premier symbole codé ne peut être que « je » dont la référence est 011. Une fois ce premier symbole décodé, on peut enlever sa référence de la suite de bits à décoder et décoder le symbole suivant : aucun symbole n'ayant pour référence 1, 10, 100 ou 10000, le deuxième symbole ne peut être que « p », dont la référence est 1000, et ainsi de suite.

Exercice 12.1

Un *indicatif téléphonique* international est un indicatif que l'on doit ajouter devant un numéro de téléphone, quand on appelle ce numéro depuis un autre pays. L'indicatif de la France est 33, celui de la Chine est 86, celui des États-Unis 1, celui de Monaco 377, etc. Expliquer pourquoi le code d'un pays n'est jamais un préfixe du code d'un autre pays.

L'utilisation de dictionnaires et de références de longueurs différentes sont les deux idées à la base des algorithmes de compression les plus courants, comme ZIP utilisé par le logiciel `gzip`.

SAVOIR-FAIRE Utiliser un logiciel de compression

Il est judicieux de compresser un fichier quand :

- ce fichier est de taille importante,
- il n'est pas déjà dans un format compressé, comme MP3,
- il comporte beaucoup de répétitions, comme une image avec de grandes zones unies,
- on n'a pas besoin d'y accéder souvent,
- on doit stocker ce fichier dans un espace limité ou le transmettre à travers un réseau à faible débit.

On peut, par exemple, utiliser le logiciel `gzip`, distribué avec Linux. On compresse un fichier `fichier.txt` avec la commande `gzip fichier.txt` ; le fichier `fichier.txt` est remplacé par le fichier `fichier.txt.gz`, que l'on décompresse avec la commande `gunzip fichier.txt.gz`.

Exercice 12.2 (avec corrigé)

Créer un fichier `a.txt` formé de la lettre « a » répétée mille fois et un fichier `alea.txt` formé de mille lettres minuscules tirées au hasard. Compresser ces deux fichiers avec le programme `gzip`. Comparer les tailles des fichiers.

Le programme suivant affiche mille lettres tirées au hasard.

```
for (i = 1; i <= 1000; i = i + 1) {
  System.out.print(Isn.asciiString ((int)
                   Math.floor(Math.random() * 26 + 97)));}
```

```
ls -l a.txt
… 1001 …
ls -l alea.txt
… 1001 …
```

```
gzip a.txt
gzip alea.txt

ls -l a.txt.gz
… 36 …
ls -l alea.txt.gz
… 652 …
```

Les deux fichiers `a.txt` et `alea.txt` ont la même taille : 1 001 octets. En revanche, le premier se compresse en un fichier de 36 octets et le second en un fichier de 652 octets.

Exercice 12.3

Créer deux fichiers PGM, contenant des images de même taille, l'une unie et l'autre aléatoire. Compresser ces deux fichiers avec le programme `gzip` et comparer les tailles des fichiers.



Exercice 12.4

On veut déterminer le type de fichiers textes sur lequel la méthode ZIP est la plus efficace.

1. La compression ZIP est-elle importante ou non sur un fichier contenant :
   - un texte littéraire,
   - un extrait d'annuaire téléphonique,
   - des caractères tapés au hasard,
   - la liste des publicités diffusées sur une chaîne de télévision pendant une journée,
   - un fichier très court : quelques caractères seulement.
2. Créer ou récupérer sur le Web de tels fichiers et tester ses prédictions à l'aide d'un logiciel de compression, par exemple `gzip`.

Compresser avec perte

Les techniques de compression des informations présentées ci-avant sont *sans perte d'informations* : en décompressant les informations compressées, on retrouve exactement les informations originales. Il existe d'autres techniques de compression dites *avec perte d'informations* : au prix d'une infime différence entre les informations originales et les informations compressées puis décompressées, on arrive à un codage moins volumineux encore.

Un exemple simple est celui d'une image entièrement blanche, à l'exception d'un pixel noir. Cette image peut être approximée par une image entièrement blanche : à l'œil nu, la différence est invisible et cette

seconde image se laisse beaucoup mieux compresser. Cette idée est utilisée dans les algorithmes de compression usuels comme JPG pour les images ou MP3 pour les sons.



Exercice 12.5

En utilisant le logiciel Gimp, ouvrir une image au format PNG, puis l'enregistrer au format JPG en qualité de 40% (avec *Enregistrer sous*). Ouvrir les deux images et les comparer à l'œil nu. Voit-on des différences ? Ouvrir ensuite en tant que calques les deux images au format JPG et PNG. Dans le mode de calques, par défaut réglé à *Normal*, choisir maintenant le mode *Différence*, qui compare les deux images en affichant la valeur absolue de la différence de la valeur de chaque pixel d'une image et de l'autre. Qu'observe-t-on ? Sachant que PNG est un format de compression d'image sans perte et que JPG est un format de compression avec perte, que peut-on en déduire ? Effectuer la même manipulation en comparant l'image au format PNG et l'image au format JPG avec une qualité 20, puis une qualité 90. Observer comment la taille du fichier JPG varie et comment la différence avec l'image au format PNG augmente ou diminue.

Exercice 12.6

Un exemple courant de compression des informations est l'utilisation de la moyenne pour compresser la suite de notes d'un élève au cours d'un trimestre. Cette compression est-elle avec ou sans perte ? Quels sont les avantages et inconvénients de cette méthode ? Proposer des méthodes de compression alternatives qui atténuent ces inconvénients.

Corriger

Les réseaux transportent de grandes quantités d'informations (des centaines de téraoctets par seconde) et les mémoires, DVD, BluRay, disques, mémoire flash, permettent souvent de stocker des téraoctets. Avec de telles quantités d'informations, il est inévitable que se produisent quelques erreurs dues au bruit sur la ligne de transmission, à des rayures d'usure sur un disque ou à des composants électroniques défaillants. C'est pourquoi des algorithmes pour détecter et corriger ces erreurs ont été inventés.

Une méthode simple pour détecter et corriger une erreur dans une suite de bits est d'y introduire une forme de *redondance*, en répétant chacun des bits plusieurs fois. Ainsi, au lieu de transmettre sur un réseau la suite de bits 10110110, on transmet la suite de bits 111000111111000111111000, où chaque bit est répété trois fois. Pour retrouver le message original, il suffit de lire les bits reçus trois par trois, en remplaçant les triplets 000 par des 0 et les triplets 111 par des 1. Si l'un des triplets reçu n'est ni 111 ni 000, par exemple si c'est 010 ou 001, on peut être certain qu'une erreur s'est glissée.

On peut même corriger l'erreur : les 0 étant majoritaires, le triplet original était sans doute 000, et l'on peut interpréter le triplet 010 ou 001 par 0. Ce code permet donc de détecter et corriger toutes les erreurs, à condition qu'il y ait au plus une erreur par triplet. En revanche, si plusieurs erreurs sont commises sur le même triplet, elles peuvent passer inaperçues ou être mal corrigées. Cette méthode fonctionne donc bien, mais elle est coûteuse car la longueur du message est triplée.

Code correcteur d'erreurs

On appelle *code correcteur d'erreurs* un code qui permet de retrouver les informations codées même en cas d'erreurs de transmission, de lecture ou d'écriture.

Dans certains cas, détecter les erreurs sans les corriger est suffisant. Par exemple, quand on transmet un message sur un réseau, la machine qui reçoit le message a souvent la possibilité de le redemander à celle qui l'a envoyé, s'il est erroné. Une manière peu coûteuse pour détecter des erreurs dans une suite de bits transmise est d'ajouter un *bit de contrôle* tous les 100 bits transmis, indiquant si le nombre de 1 dans ce paquet de 100 bits est pair (0) ou impair (1). La longueur des messages est ainsi augmentée de 1 % seulement. Si une erreur se produit lors de la transmission des 101 bits, c'est-à-dire si un 0 est remplacé par un 1, ou un 1 par un 0, la parité du nombre de 1 est changée et l'erreur est détectée. En revanche, si deux erreurs se produisent dans la même suite de 101 bits, elles passeront inaperçues.

ALLER PLUS LOIN **Longueur des suites**

Plutôt que des suites de 100 bits, on peut utiliser des suites de 10 bits ou de 1 000 bits. Plus la suite est longue, plus la méthode est économe, mais plus la probabilité de voir deux erreurs se produire dans la même suite, et donc passer inaperçues, est élevée.

Dans d'autres cas, il est nécessaire de corriger les erreurs. Par exemple, quand on lit un DVD et que l'on détecte une erreur, on veut pouvoir la corriger au vol et non demander au spectateur d'aller acheter un autre DVD pour voir la fin du film. Une méthode de correction des erreurs moins coûteuse que le triplement des bits décrit ci-avant consiste à ajouter seulement 20 bits de contrôle tous les 100 bits, de la manière suivante : on organise le paquet de 100 bits en un tableau de 10 lignes sur 10 colonnes et on ajoute un bit de contrôle par ligne et un bit de contrôle par colonne, soit 20 bits au total. Ce bit indique simplement si le nombre de 1 dans la ligne ou la colonne est pair ou impair. Quand on reçoit le message, si on détecte une erreur dans la ligne l et une erreur dans la colonne c, on sait que le bit erroné est celui qui se trouve dans le tableau à la ligne l et à la colonne c ; il suffit, pour corriger le message, de remplacer ce bit par un 1 si c'est un 0 ou par un 0 si c'est un 1. Si on détecte une erreur dans une ligne, mais aucune erreur dans les colonnes, ou le contraire, c'est que l'erreur porte sur le bit de contrôle lui-même et il n'y a donc rien à corriger dans le message. Cette méthode demande donc d'allonger le message de 20 % et elle permet de corriger toutes les erreurs à condition qu'une erreur au plus se produise dans chaque suite de 120 bits.



Exercice 12.7

On utilise la méthode décrite précédemment pour transmettre 16 bits. Par exemple, pour transmettre le message 0011010111010111 on construit le tableau :

					Colonne de contrôle
	0	0	1	1	0
	0	1	0	1	0
	1	1	0	1	1
	0	1	1	1	1
Ligne de contrôle	1	1	0	0	

et on transmet la suite 0011010111010111 0011 1100.

De combien de bits de contrôle a-t-on besoin ?

Montrer que si on reçoit le message 00?10?0111010?11 001?1100, où les « ? » représentent des bits inintelligibles, il est possible de reconstituer entièrement les données qui ont été envoyées, y compris les bits de contrôle.

Est-il possible de reconstituer le message original si on reçoit la séquence suivante : 0??10?0111010?11001?1100 ?

Montrer que le message suivant : 101001111001001000010100, transmis suivant la même méthode, est incohérent. Expliquer cette incohérence. Comment y remédier ?

Chiffrer

Sûreté et sécurité

Dans la section précédente, nous avons présenté des méthodes qui permettent de protéger les informations contre des erreurs accidentelles. Cela s'appelle augmenter la *sûreté des informations*, c'est-à-dire la protéger des erreurs involontaires, telles les erreurs de transmission. À l'inverse, on cherche dans cette section à augmenter la *sécurité des informations*, c'est-à-dire à la protéger contre l'action de personnes malveillantes.

Chiffrer

On appelle *chiffrer* des informations le fait de les exprimer sous la forme d'une suite de bits, avec un code choisi pour rendre cette suite aussi inintelligible que possible, sauf pour son destinataire.

Une méthode pour protéger les informations contre les actions d'une personne malveillante qui veut y accéder alors qu'elles ne lui sont pas destinées consiste à les chiffrer.

Cette idée de chiffrement est ancienne puisqu'on sait que Jules César avait déjà mis au point un algorithme pour transmettre des ordres à ses armées de manière secrète. Pour cela, il utilisait le *code de César*, qui consistait à remplacer chaque lettre d'un message par celle située trois lettres plus loin dans l'alphabet : les « a » étaient replacés par des « d », les « b » par des « e », etc. Ainsi « Veni, vidi, vici » se chiffrait en « Zhqm, zmgm, zmfm » – attention, il n'y a pas de « j », de « u », ni de « w » en latin.

Cette méthode n'est en fait pas très bonne. D'une part, une fois qu'on la connaît, on peut déchiffrer tous les messages très simplement. D'autre part, même si on ne connaît pas la correspondance entre les lettres, celle-ci est facile à deviner, car la fréquence des lettres, dans une langue donnée, est à peu près constante d'un texte à un autre. Ainsi, si le mes-

sage est en français, il suffit de remarquer que la lettre « h » est la plus fréquente dans les messages chiffrés pour deviner que cette lettre est le code de la lettre « e », lettre la plus fréquente en français.

Des méthodes plus robustes ont donc été développées. Une d'elles est *la méthode du masque jetable*, dans laquelle les deux interlocuteurs se mettent d'accord, avant d'échanger un message, sur une *clé* qui définit les bits du message que l'expéditeur laissera identiques et ceux qu'il inversera, c'est-à-dire remplacera par un 1 si c'est un 0 ou par un 0 si c'est un 1. Par exemple, pour échanger un message long de huit bits, les interlocuteurs se mettent d'accord sur le fait que l'expéditeur le chiffrera en inversant le premier, le deuxième, le troisième, le septième et le huitième bit et en laissant les autres inchangés. Ainsi le message 01101101 sera chiffré en 10001110. Le récepteur du message, connaissant lui aussi la clé, n'aura qu'à inverser les mêmes bits pour retrouver le message original : 01101101.

La clé qui indique quels bits laisser en l'état et quels bits inverser est elle-même exprimée par une suite de bits : 0 signifiant qu'on laisse le bit inchangé et 1 indiquant qu'on l'inverse. Ainsi, la clé ci-avant s'exprime par la suite de bits : 11100011. Une clé, dans cette méthode, s'appelle aussi un *masque*. Chiffrer le message consiste à effectuer un ou exclusif, bit à bit, entre le message et le masque. En effet, effectuer un ou exclusif entre un bit du message et un 0 du masque laisse ce bit inchangé (0 *oux* 0 = 0, 1 *oux* 0 = 1) et effectuer un ou exclusif entre un bit du message et un 1 du masque inverse ce bit (0 *oux* 1 = 1, 1 *oux* 1 = 0). Cette même opération permet de retrouver le message original à partir du message chiffré.

Cette méthode a été employée par les diplomates et les services secrets depuis le début du XXe siècle, mais elle présente plusieurs inconvénients. Le premier est que le masque doit avoir la même longueur que le message lui-même, il doit être complètement aléatoire et il ne doit jamais être réutilisé, d'où le nom de *masque jetable*. Cette méthode demande donc de fabriquer des masques aléatoires de très grande taille, ce qui est plus difficile qu'il n'y paraît. Un second problème est que ce masque doit être secrètement échangé entre les interlocuteurs, avant la transmission du message, ce qui est difficile. Si une personne malveillante arrive à intercepter un masque au cours de cet échange, elle pourra déchiffrer les messages échangés.

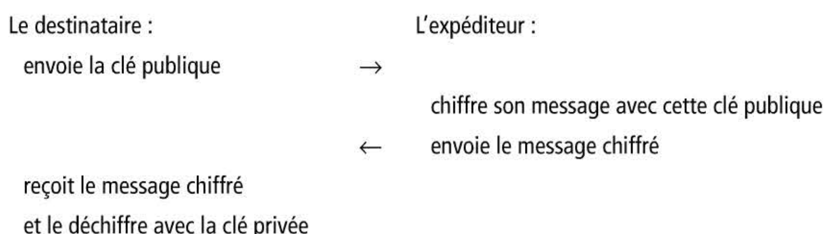
Cela a mené à la conception de méthodes alternatives qui ne reposent pas sur l'échange préalable d'une clé secrète entre les interlocuteurs : les méthodes *à clé publique – clé privée*.

Une métaphore aide à saisir le principe de ces méthodes : deux personnes, l'*expéditeur* et le *destinataire*, souhaitent échanger un message confidentiel, mais elles n'ont pas la possibilité de se mettre d'accord au préalable sur une clé secrète. Une possibilité est que le destinataire envoie par poste à l'expéditeur un cadenas ouvert dont il garde la clé.

L'expéditeur met son message dans une boîte, qu'il ferme avec le cadenas reçu, et envoie cette boîte au destinataire, qui n'a plus qu'à ouvrir le cadenas avec la clé qu'il a gardée. À aucun moment, il ne s'est séparé de la clé, si bien que ni le facteur, ni personne à l'exception du destinataire, n'a le moyen d'ouvrir la boîte. Le message reste donc confidentiel, si l'on est certain que le cadenas utilisé est bien celui du destinataire.

Le fait de pouvoir fermer un cadenas sans avoir la clé qui permet de l'ouvrir est ce qui permet à cette méthode de fonctionner. Les méthodes à clé publique – clé privée reposent sur un mécanisme similaire : la possibilité de chiffrer un message sans disposer de la clé qui permet de le déchiffrer.

Une telle méthode recourt à deux clés : une *clé publique*, diffusée à tous par le destinataire pour le chiffrement des messages, et une *clé privée*, qu'il garde secrète, permettant de les déchiffrer. On peut schématiser ce mécanisme ainsi :



La méthode à clé publique – clé privée la plus utilisée est la méthode RSA, qui doit son nom à celui de ses inventeurs : Rivest, Shamir et Adleman. Dans cette méthode, la clé privée est formées de trois nombres, d, e et n tels que pour tout entier w inférieur à n, $(w^e \% n)^d \% n = w$, où l'opération % est le reste de la division euclidienne. La clé publique correspondante est formée des nombres e et n uniquement.

Un message à transmettre est d'abord exprimé sous la forme d'une suite de bits, que l'on interprète comme un nombre entier w, inférieur à n. L'expéditeur le chiffre en $w' = w^e \% n$, puis envoie ce message chiffré w' au destinataire, qui le déchiffre en calculant $w'^d \% n$, qui est donc égal à $(w^e \% n)^d \% n$, c'est-à-dire à w, le message original. Pour chiffrer un message, l'expéditeur n'a pas besoin de connaître la clé privée, car le nombre d n'est utilisé que dans la phase de déchiffrement.

Si une personne malveillante a accès aux nombres e et n et à un message chiffré w', elle peut essayer de déduire d de e et n ou directement trouver un nombre w tel que $w^e \% n = w'$. L'un et l'autre de ces calculs sont possibles, mais ils sont très longs, pour peu que e et n soient assez grands. Les méthodes les plus rapides que l'on connaisse aujourd'hui demandent plusieurs années de calcul, quand n est de l'ordre de quelques milliers de chiffres binaires.

Les méthodes à clé publique – clé privée sont donc, en théorie, d'un niveau de sécurité inférieur aux méthodes à clés secrètes : quand on connaît la méthode de chiffrement et un message chiffré w', on peut, en théorie, essayer de chiffrer tous les messages w possibles, jusqu'à en trouver un qui se code en w'. Toutefois, cela demande en pratique un temps de calcul énorme. C'est donc un inconvénient négligeable à coté de l'avantage que présente le fait de ne pas avoir besoin d'échanger secrètement une clé.

ALLER PLUS LOIN **Construire une clé RSA**

Pour construire une clé RSA, il suffit de choisir deux nombres premiers et distincts p et q, par exemple $p = 3017642249$ et $q = 6644055791$. On choisit ensuite un nombre d premier avec $(p - 1)(q - 1)$, par exemple $d = 2596516757$ et un nombre e tel que $(e\ d)$ % $((p - 1)(q - 1)) = 1$, par exemple $e = 3566116940332599833$. On pose ensuite $n = pq$, dans cet exemple $n = 20049383459634713959$. La clé privée est formée des nombres, d, e et n et la clé publique des nombres e et n uniquement.

On démontre alors que pour tout entier w inférieur à n, $(w^e$ % $n)^d$ % $n = w$. Pour cela, on montre que $w^{ed} - w$ est un multiple de p, en utilisant le petit théorème de Fermat : si p est un nombre premier et w un nombre qui n'est pas un multiple de p alors w^{p-1} % $p = 1$. La démonstration ne demande que quelques lignes. On montre de même que $w^{ed} - w$ est un multiple de q. Comme p et q sont deux nombres premiers et différents, $w^{ed} - w$ est un multiple de $n = pq$. Donc w^{ed} % $n = w$, c'est-à-dire $(w^e$ % $n)^d$ % $n = w$.

La méthode connue la plus rapide pour déchiffrer un message, quand on ne connaît pas la clé privée, consiste à la déduire de la clé publique en factorisant le nombre n en un produit de deux nombres premiers, on obtient ainsi p et q, donc $(p - 1)(q - 1)$, puis en connaissant $(p - 1)(q - 1)$ et e, on peut retrouver d. Toutefois, factoriser un nombre de quelques milliers de chiffres binaires demande plusieurs années de calcul.

ALLER PLUS LOIN **Authentifier**

Les méthodes à clé publique – clé privée permettent aussi à une autorité d'*authentifier* un utilisateur, c'est-à-dire de vérifier son identité. Pour cela, il lui suffit de détenir la clé publique de la personne à authentifier et de vérifier qu'elle détient bien sa clé privée en lui faisant décoder un message.



On voit donc ici que seul un utilisateur qui détient la clé privée peut déchiffrer le message de test, ce qui garantit l'authentification, si on fait l'hypothèse que la clé publique utilisée est bien celle de l'utilisateur, et non celle d'un imposteur.

Exercice 12.8

Utiliser la méthode du masque jetable pour transmettre un nombre à 4 chiffres à son voisin.



Exercice 12.9

Que se passerait-il si on découvrait un algorithme rapide pour factoriser un nombre entier en un produit de nombres premiers ?



Exercice 12.10

Pour retirer de l'argent avec une carte de retrait, on doit entrer un code secret compris entre 0000 et 9999 et la carte est avalée au bout de trois erreurs : quelle est la probabilité de réussir à utiliser une carte sans en avoir le code ? Et si on pouvait, au bout de deux essais, aller essayer la carte dans un autre distributeur ?



Exercice 12.11

Pour retirer de l'argent avec une carte de retrait, on doit entrer un code secret compris entre 0000 et 9999. On suppose que l'on peut essayer autant de codes que l'on veut, mais avec un délai de 1 s entre le premier et le deuxième essai, 10 s entre le deuxième et le troisième, 100 s entre le troisième et le quatrième, etc. Au bout de combien de temps est-on sûr de pouvoir trouver le code ? Quel est le temps moyen pour le trouver ? Cette méthode est utilisée pour protéger les serveurs contre les attaques par des logiciels qui testent tous les mots de passe possibles pour s'authentifier.



Exercice 12.12

Expliquer la méthode d'authentification présentée dans l'encadré, avec la métaphore du cadenas, qui fait office de clé publique, et sa clé privée.



Exercice 12.13

Dans un univers imaginaire, Aïcha est la seule personne à savoir factoriser l'expression $x^2 + a\,x + b$, mais tout le monde sait développer l'expression $(x - u)\,(x - v)$. Imaginer une méthode qui permet à Aïcha de recevoir d'une autre personne des messages chiffrés. Cette méthode permet-elle à Aïcha d'envoyer des messages chiffrés ? Permet-elle l'authentification d'Aïcha ou des autres personnes de ce monde imaginaire ?

Ai-je bien compris ?

- Qu'est-ce que compresser des informations ?
- Qu'est-ce qu'un code correcteur d'erreurs ?
- Qu'est-ce que chiffrer des informations ?

Troisième partie

Machines

Dans cette troisième partie, nous voyons que derrière les informations, il y a toujours des objets matériels : ordinateurs, réseaux, robots, etc. Les premiers ingrédients de ces machines sont des portes booléennes (chapitre 13) qui réalisent les fonctions booléennes vues au chapitre 10. Ces portes demandent à être complétées par d'autres circuits, comme les mémoires et les horloges, qui introduisent une dimension temporelle (chapitre 14). Nous découvrons comment fonctionnent ces machines que nous utilisons tous les jours (chapitre 15). Nous verrons que les réseaux, comme les oignons, s'organisent en couches (chapitre 16*). Et nous découvrons enfin les entrailles des robots, que nous apprenons à commander (chapitre 17*).


SALUT ! AU FAIT, TU AS REÇU UN APPEL DES ANNÉES 1970.
AH BON ? ILS VOULAIENT QUOI ?


JE SAIS PAS, ILS N'ONT PAS LAISSÉ DE MESSAGE.
BIZARRE...


1974:
POUR LAISSER UN MESSAGE, APPUYER SUR LA TOUCHE 1.

XKCD

Les portes booléennes

13



Frances Allen (1932-) est une pionnière de la parallélisation automatique des programmes, c'est-à-dire de la transformation de programmes destinés à être exécutés sur un ordinateur séquentiel – contenant un unique processeur – en des programmes destinés à être utilisés sur un ordinateur parallèle – contenant plusieurs processeurs. Elle est aussi à l'origine de nouvelles méthodes, fondées sur la théorie des graphes, pour optimiser les programmes. Elle a reçu le prix Turing en 2006 pour ces travaux.

Au commencement était le transistor,
puis nous créâmes les portes booléennes
et, à la fin de la journée, les ordinateurs.

Dans ce chapitre, nous voyons de quoi sont faits les ordinateurs à l'échelle microscopique. Nous partons du transistor et construisons successivement des circuits *non* et *ou* qui vont nous permettre ensuite de construire les circuits de toutes les fonctions booléennes, comme nous l'avons vu au chapitre 10.

Nous connaissons des algorithmes depuis plus de quatre mille ans, pourtant nous n'avons pas cherché à les exprimer dans des langages de programmation avant le milieu du XX[e] siècle. C'est en effet seulement à ce moment que les progrès de l'électronique nous ont permis de construire les premiers ordinateurs. La construction de ces machines a donc eu un effet important sur la manière dont nous concevons aujourd'hui les notions d'algorithme, de langage et d'information.

Le circuit NON

ALLER PLUS LOIN

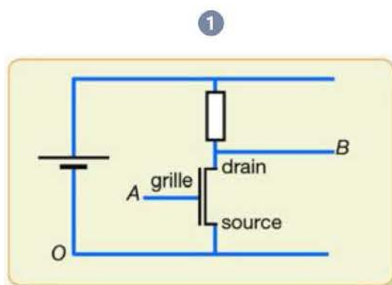
Les circuits CMOS

Dans ce livre nous utilisons un seul type de transistors appelés N-Mos. On construit aujourd'hui plus souvent des circuits qui utilisent deux types de transistors N-Mos et P-Mos, afin de minimiser la consommation d'électricité et la production de chaleur.

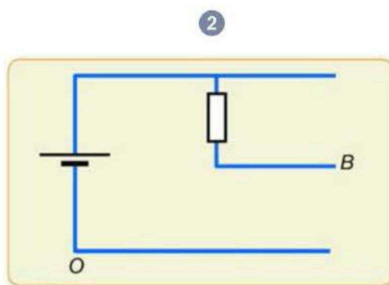
Comme beaucoup de systèmes complexes, un ordinateur peut se décrire à de nombreuses échelles. À l'échelle la plus petite, un ordinateur est un assemblage de transistors. Un transistor est un circuit électronique à trois fils appelés *le drain*, *la source* et *la grille*. La résistance entre le drain et la source est ou bien très petite ou bien très grande en fonction de la tension appliquée entre la grille et la source. Quand cette tension est inférieure à un certain seuil, cette résistance est très grande, on dit que le transistor est *bloqué* ; quand la tension est supérieure à ce seuil, la résistance est très petite, on dit que le transistor est *passant*. Avec un transistor, une résistance et un générateur dont la tension est supérieure au seuil de basculement du transistor, on peut construire le circuit ❶.

Si on applique entre le point A et le point O une tension inférieure au seuil de basculement du transistor, celui-ci est bloqué et le circuit est équivalent au circuit ❷, si bien que la tension entre les points B et O est égale à la tension d'alimentation. Elle est donc supérieure au seuil de basculement. Si, en revanche, on applique entre les points A et O une tension supérieure au seuil de basculement du transistor, celui-ci est passant et le circuit est équivalent au circuit ❸, si bien que la tension entre les points B et O est nulle. Elle est donc inférieure au seuil de basculement.

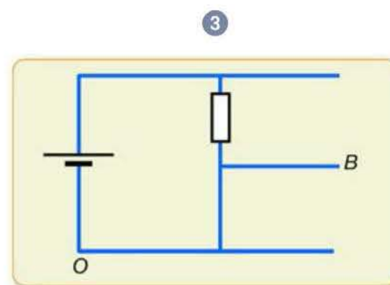
❶



❷



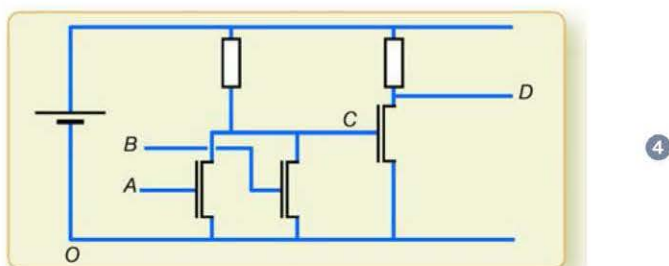
❸



Si on décide qu'une tension inférieure au seuil de basculement représente le bit 0 et qu'une tension supérieure à ce seuil représente le bit 1, les deux remarques précédentes se reformulent ainsi : si on donne au circuit le bit 0 en *A*, il donne le bit 1 en *B* ; si on lui donne le bit 1 en *A*, il donne le bit 0 en *B*. Autrement dit, ce circuit calcule une fonction booléenne : la fonction *non*.

Le circuit OU

Le circuit ④ est construit selon les mêmes principes, mais il a deux entrées *A* et *B*.



④

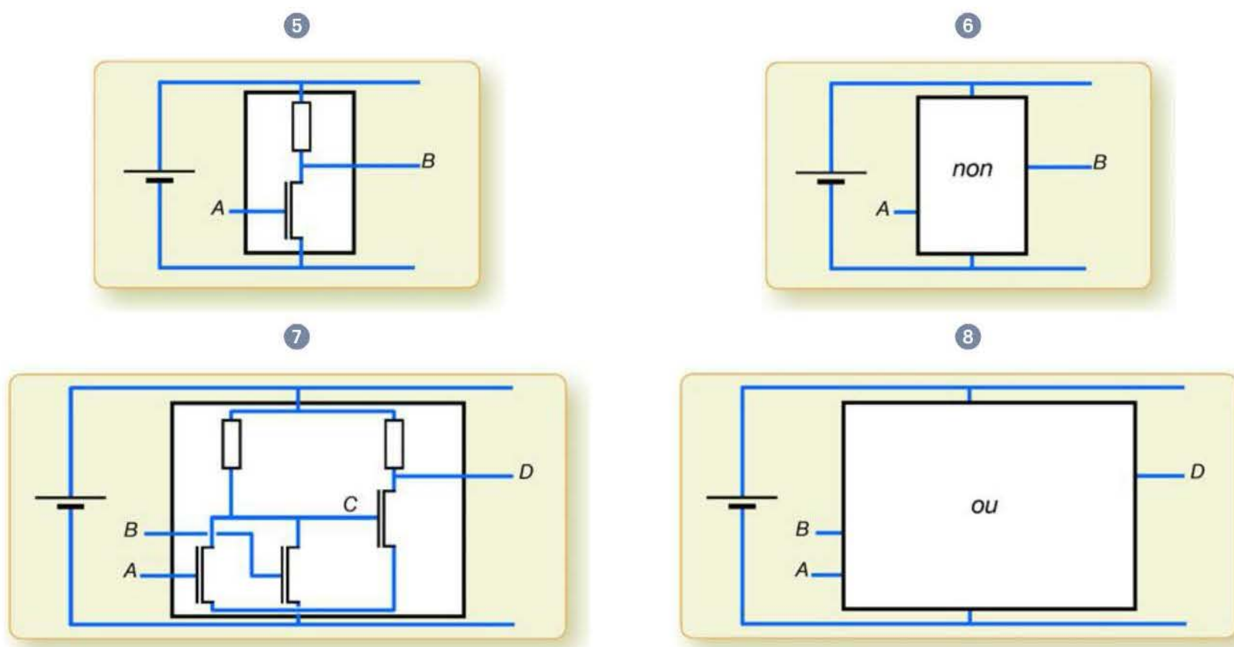
Si on donne aux deux entrées *A* et *B* le bit 0, les deux transistors dans la partie gauche du circuit sont bloqués, si bien que la tension entre les points *C* et *O* est égale à la tension d'alimentation, supérieure au seuil de basculement. Le transistor de droite est donc passant et la tension entre les points *D* et *O* est nulle ; autrement dit le point *D* est dans l'état 0.

Si on donne à l'une ou l'autre des entrées *A* et *B* le bit 1, au moins l'un des deux transistors dans la partie gauche du circuit est passant, si bien que la tension entre les points *C* et *O* est nulle. Le transistor de droite est donc bloqué et la tension entre *D* et *O* est égale à la tension d'alimentation. Le point *D* est par conséquent dans l'état 1.

A	B	D
0	0	0
0	1	1
1	0	1
1	1	1

La table de ce circuit est donc la suivante (voir ci-contre) où l'on reconnaît la table de la fonction *ou*.

On peut schématiser ces circuits de manière plus succincte en remplaçant le morceau de dessin représentant le transistor et la résistance encadrés dans la figure ⑤ par un simple rectangle (⑥) et en remplaçant de même le morceau de dessin représentant les trois transistors et les deux résistances encadrés dans la figure ⑦ par un rectangle (⑧).



On arrive ainsi à une représentation, à un autre niveau, du même circuit.

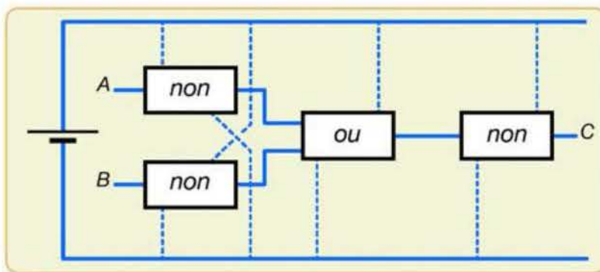
Quelques autres portes booléennes

Les circuits *non* et *ou* s'appellent des *portes booléennes* ou parfois des *portes logiques*.

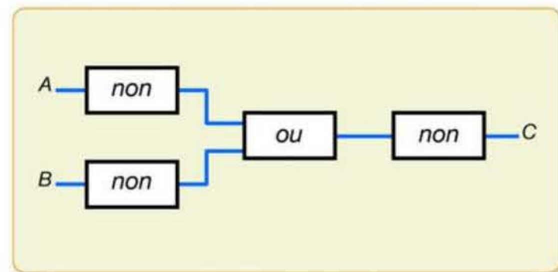
Dans ce chapitre et le suivant, on constitue petit à petit une boîte à outils de circuits réutilisables pour concevoir des circuits plus sophistiqués. Les portes *non* et *ou* sont les deux premiers éléments de cette boîte à outils.

Bien souvent, quand on représente un circuit, on ne dessine pas le générateur : il est implicite que chaque porte est alimentée. On obtient alors une troisième manière de représenter les circuits où le circuit 9 est représenté comme sur le schéma 10.

9

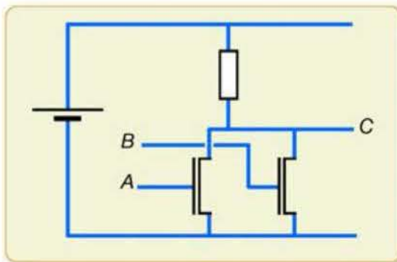


10



Exercice 13.1 (avec corrigé)

Quelle est la table du circuit suivant ?



Si on donne à l'une ou l'autre des entrées A *et* B *le bit 1, au moins l'un des deux transistors dans la partie gauche du circuit est passant, si bien que le point* C *est dans l'état 0. Sinon le point* C *est dans l'état 1.*

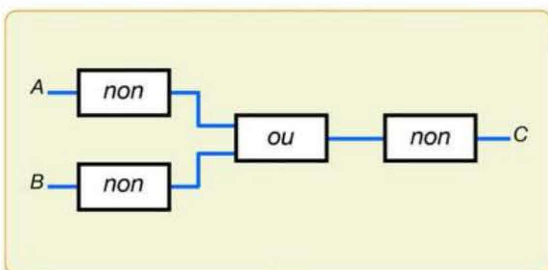
La table de ce circuit est donc la suivante (voir ci-contre).

Il s'agit de la table de la fonction booléenne qui à A *et* B *associe* non (A ou B)*.*

A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Exercice 13.2 (avec corrigé)

Quelle est la table du circuit suivant ? Est-ce la table d'une fonction booléenne connue ?



La table de ce circuit est :

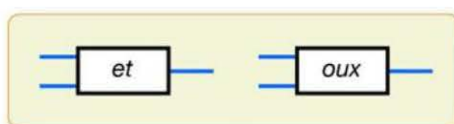
A	B	C
0	*0*	*0*
0	*1*	*0*
1	*0*	*0*
1	*1*	*1*

C'est celle de la fonction booléenne et.

Exercice 13.3

Construire un circuit réalisant la fonction booléenne *ou exclusif*, vue au chapitre 10.

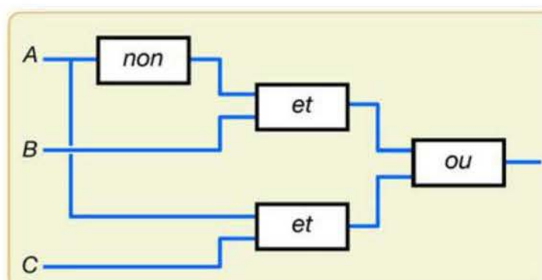
En plus des portes *ou* et *non*, on a désormais dans sa boîte à outils les portes *et* et *oux* :



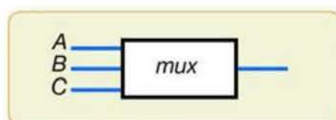
Exercice 13.4 (avec corrigé)

Construire un circuit réalisant la fonction multiplexeur vue au chapitre 10.

La fonction multiplexeur peut se définir par mux (A, B, C) = (non (A) et B) ou (A et C)*. Un circuit, parmi d'autres, réalisant cette fonction est donc :*



On peut désormais utiliser directement le circuit suivant, dont l'unique sortie transmet la valeur de B *ou de* C *selon la valeur de* A *:*



Exercice 13.5 (avec corrigé)

Construire le circuit réalisant le calcul de la fonction *Cout* définie par la table :

A	B	Cin	Cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Quelle est cette fonction ?

On utilise la méthode de décomposition par multiplexage (voir le chapitre 10). La fonction Cout (Cin, A, B*) s'écrit* mux (Cin, g (A, B), g' (A, B)), *avec* g (A, B) *et* g' (A, B) *définies par les tables ci-contre :*

g (A, B)

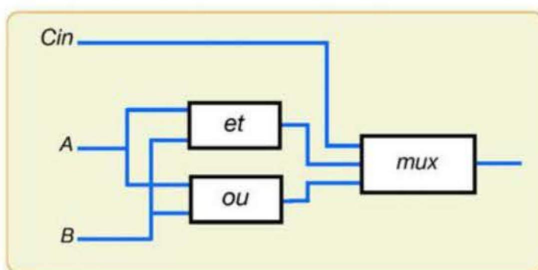
A	B	g
0	0	0
0	1	0
1	0	0
1	1	1

g' (A, B)

A	B	g'
0	0	0
0	1	1
1	0	1
1	1	1

On reconnaît les tables des fonctions et *et* ou, *respectivement. Si bien que* Cout (A, B , Cin*)* = mux (Cin, A et B, A ou B*).*

Un circuit calculant cette fonction est donc :



Cette fonction est la fonction chiffre des deuzaines de A + B + Cin, *qui sert au calcul de la retenue dans l'algorithme de l'addition (voir le chapitre 18). Ce circuit porte le nom de* Carry out *(retenue sortante).*



Exercice 13.6

Construire un circuit réalisant les opérations calculant la fonction *chiffre des unités de A + B + Cin* (voir le chapitre 18). Construire un circuit *additionneur un bit* qui prend en entrée deux nombres de un bit et donne en sortie leur somme, sur deux bits.

Construire un circuit à quatre entrées et trois sorties, qui ajoute deux nombres exprimés sur deux bits.

Construire un circuit à seize entrées et neuf sorties qui ajoute deux nombres binaires de huit bits.



Exercice 13.7

Construire un circuit à 9 entrées A_0 … A_7 et D et 8 sorties B_0 … B_7 telles que :

1. lorsque $D = 0$, $B_i = A_i$ pour i compris entre 0 et 7,
2. lorsque $D = 1$, $B_i = A_{i-1}$ pour i compris entre 1 et 7 et $B_0 = 0$.

Quand $D = 1$, ce circuit réalise un *décalage à gauche* d'un nombre exprimé en binaire sur huit bits, ce qui correspond à la multiplication par 2 de ce nombre. Dans cette multiplication, le chiffre le plus à gauche, A_7, est oublié afin de faire tenir le résultat sur huit bits.



Exercice 13.8

Construire un circuit à 11 entrées A_0 … A_7 et D_0 … D_2 et 8 sorties B_0 … B_7 telles que $B_i = A_{i-d}$ pour i entre d et 7 et $B_i = 0$ pour i entre 0 et (d - 1), où d est le nombre entre 0 et 7 représenté en binaire par D_0 … D_2. Ce circuit réalise un *décalage à gauche* de d bits d'un nombre binaire de huit bits, ce qui correspond à la multiplication par 2^d. Ce circuit est un composant important d'un circuit réalisant une multiplication binaire.

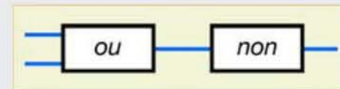
À l'issue de ce chapitre, on dispose donc, parmi d'autres, des circuits booléens suivants dans sa boîte à outils :

- les portes booléennes *non*, *ou*, *et* et *oux*,
- le multiplexeur *mux*,
- des additionneurs de nombres de différentes tailles,
- des multiplieurs par 2 et par 2^d.

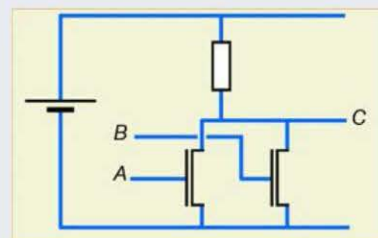
ALLER PLUS LOIN **Penser les systèmes complexes**

Au cours de ce chapitre, nous sommes partis d'une manière de décrire des circuits formés de transistors et de résistances pour, peu à peu, faire émerger une autre manière de décrire certains de ces circuits, sous la forme d'assemblages de portes booléennes. Comme chaque porte est constituée de plusieurs composants électroniques, cette description sous la forme d'un assemblage de portes booléennes est une description *à plus grande échelle* que celle sous forme de transistors et de résistances. Une question se pose alors : peut-on concevoir des circuits en assemblant des portes booléennes et en ignorant complètement la manière dont ces portes sont réalisées avec des transistors ?

Cette question demande une réponse nuancée : comprendre à la fois l'échelle des portes booléennes et celle des transistors permet parfois de réaliser des circuits plus petits, donc moins chers et plus rapides. Par exemple, si on veut construire un circuit qui réalise l'opération booléenne qui à A et B associe *non* (*A ou B*) et si l'on sait uniquement associer des portes booléennes, on construira le circuit suivant



qui, *in fine*, utilise quatre transistors, trois pour la porte *ou* et un pour la porte *non*. Si, en revanche, on sait aussi comment ces portes sont construites avec des transistors, on peut remarquer que le circuit



qui ne comporte que deux transistors, convient. Raisonner à une petite échelle permet donc d'économiser deux transistors. Il est souvent utile, quand on raisonne à une échelle donnée, de faire une incursion à l'échelle inférieure ou à l'échelle supérieure.

Cela dit, il y a aussi des avantages à construire des circuits avec des portes booléennes en ignorant, ou en feignant d'ignorer, les échelles inférieures. Par exemple, les portes booléennes sont aujourd'hui fabriquées avec des transistors, mais par le passé, elles ont été fabriquées avec d'autres composants : des relais, des tubes à vide, etc. et il est possible que, dans le futur, elles soient réalisées avec d'autres composants, aujourd'hui non encore inventés. Raisonner à l'échelle des portes booléennes, sans prendre en compte la manière dont elles sont fabriquées, permet de conserver la même organisation des circuits à l'échelle des portes, même quand la manière dont ces portes sont fabriquées change.

De plus, il est illusoire d'espérer penser simultanément un système aussi complexe qu'un ordinateur à toutes les échelles. S'il est donc, bien entendu, utile d'avoir une culture générale qui donne une idée de la manière dont un ordinateur se décrit à toutes les échelles, il est aussi souvent nécessaire de savoir penser à une échelle unique, en ignorant, ou en feignant d'ignorer, la manière dont les composants que l'on assemble sont fabriqués. Ainsi, aux chapitres 14 et 15, on construira des circuits de plus en plus élaborés en réutilisant les circuits précédents comme de nouveaux composants.

ALLER PLUS LOIN **Qui a inventé l'ordinateur ?**

Contrairement à la pénicilline, il est très difficile de dire qui a inventé l'ordinateur.

Sans remonter aux machines à calculer du XVIIe siècle de Wilhelm Schickard, Blaise Pascal, Gottfried Wilhelm Leibniz, etc., ou à la machine analytique imaginée au XIXe siècle par Charles Babbage, l'apparition de l'ordinateur a été préparée par une grande créativité dans le domaine de la construction de machines à la fin du XIXe siècle et au début du XXe siècle, avec, par exemple, la machine à recensement de Herman Hollerith construite en 1889 ou l'analyseur différentiel de Harold Locke Hazen et Vannevar Bush construit entre 1928 et 1931, qui étaient déjà des machines polyvalentes.

La notion d'universalité a été définie mathématiquement en 1936 par Alonzo Church et Alan Turing et il semble que la première machine universelle ait été le Z3, construite en 1941 par Konrad Zuse, même si on ne s'en est rendu compte qu'*a posteriori*. La première machine électronique à utiliser le binaire semble avoir été la machine Colossus, construite par Thomas Flowers au cours de la seconde guerre mondiale, mais cette machine n'était pas universelle. La première machine conçue pour être universelle fut sans doute l'ENIAC, construite en 1946 par John Mauchly, Presper Eckert et John Von Neumann. Pour certains néanmoins, ce n'était pas encore un ordinateur, son programme n'étant pas enregistré dans la mémoire. La première machine à programme enregistré a sans doute été la machine *Baby*, construite à Manchester en 1948 par Frederic Williams, Tom Kilburn et Geoff Tootill.

Il semble donc difficile d'attribuer l'invention de l'ordinateur à un inventeur unique. Il y a plutôt eu un foisonnement d'innovations de la fin des années trente au début des années cinquante qui, chacune à sa manière, ont contribué à l'invention de l'ordinateur.

Ai-je bien compris ?

- Comment réaliser une porte *non* avec des transistors ?
- Comment réaliser une porte *ou* avec des transistors ?
- Est-il nécessaire de savoir réaliser une porte *non* et une porte *ou* avec des transistors pour les assembler en des circuits plus complexes ?

Le temps et la mémoire

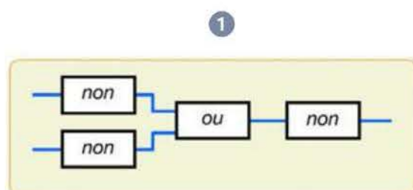
14

Le temps est ce qui permet d'éviter de tout faire en même temps.

Dans ce chapitre, nous voyons comment les circuits électroniques prennent le temps en compte. Nous voyons d'abord comment fabriquer un circuit mémoire. Puis, comment un circuit particulier, l'horloge, permet de synchroniser tous les autres.



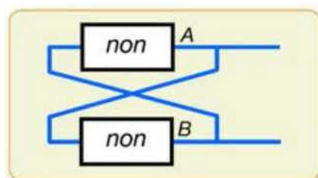
Otto Schmitt (1913-1998) est un pionnier du *génie biomédical*. En 1934, en étudiant la propagation de l'influx nerveux dans les nerfs des calmars, il a compris qu'un circuit en boucle fermée positive – c'est-à-dire dans lequel la sortie est connectée à l'entrée, sans inversion de valeur – avait deux états stables et pouvait donc être utilisé pour mémoriser une grandeur. En électronique, une *bascule de Schmitt* est une forme de circuit bistable, qui utilise cette idée de boucle fermée positive.



Les circuits que nous avons vus au chapitre 13, par exemple le circuit 1 illustré ci-contre, ont des entrées (deux à gauche sur la figure) et des sorties (une à droite sur la figure) et l'état des sorties est déterminé par celui des entrées. Dans cet exemple, la sortie est dans l'état 1 quand les deux entrées sont dans l'état 1 et elle est dans l'état 0 quand au moins l'une des entrées est dans l'état 0. L'état des sorties à un instant donné dépend de l'état des entrées à ce même instant, mais pas de l'état des entrées une seconde ou une minute plus tôt. Un tel circuit, qui ignore le temps, s'appelle un circuit *combinatoire*. Il y a, autour de nous, beaucoup de circuits combinatoires. Par exemple, une lampe est allumée quand son interrupteur est fermé et elle est éteinte quand cet interrupteur est ouvert ; l'état de la lampe dépend de la position de l'interrupteur, mais pas de la position de l'interrupteur une seconde ou une minute plus tôt.

Cependant, il y a aussi autour de nous des circuits moins amnésiques, dont l'état à un instant donné dépend non seulement de l'état de ses entrées à cet instant, mais aussi de leur état passé. Par exemple, quand nous appuyons sur la touche 1 d'une calculatrice, le chiffre 1 apparaît sur l'écran, mais quand nous relâchons cette touche, le chiffre 1 ne disparaît pas : l'état de l'écran à un instant donné dépend donc non seulement de l'état du clavier à ce même instant, mais aussi de toute l'histoire du clavier. Un tel circuit s'appelle un *circuit séquentiel*. Les ordinateurs sont, bien entendu, des circuits séquentiels car, comme nous l'avons vu au chapitre 1, l'exécution d'un programme modifie un état, qui est une description abstraite de l'état de l'ordinateur et dépend donc de toutes les instructions exécutées dans le passé.

La mémoire



Le circuit séquentiel le plus simple est *le circuit mémoire un bit* qui permet de mémoriser un 0 ou un 1. Construire un tel circuit n'est pas difficile, mais il faut procéder en plusieurs étapes. La première est de construire un circuit qui a deux états stables, par exemple celui de la figure ci-contre.

Ce circuit a deux états stables car :

- Si la sortie *A* de la première porte *non* est dans l'état 0, alors l'entrée de la seconde porte *non*, qui est *A* également, est aussi dans l'état 0 ; par conséquent, sa sortie *B* est dans l'état 1, donc l'entrée de la première porte, qui est *B* également, est dans l'état 1, ce qui participe à perpétuer le fait que sa sortie *A* soit dans l'état 0.

ALLER PLUS LOIN **La rupture de symétrie**

Il est difficile de prédire l'état dans lequel se retrouve un circuit mémoire un bit quand on le met sous tension. Il se trouve d'abord, pendant une courte durée, dans un état instable dans lequel les deux sorties *A* et *B* sont dans l'état 0. L'entrée des deux portes *non* est alors dans l'état 0. Rapidement, l'une d'elles produit un état 1 en sortie, un peu avant l'autre, si bien que l'autre, ayant désormais son entrée dans l'état 1, garde sa sortie dans l'état 0. C'est donc une différence de vitesse entre les portes *non* qui détermine l'état du circuit quand on le met sous tension. Cette différence de vitesse est elle-même due à une infime différence de température, de longueur de fil, de pureté des matériaux utilisés pour construire les transistors, etc. Ce phénomène est un exemple de *rupture de symétrie*. Dans le circuit, les points *A* et *B* sont parfaitement symétriques, mais pour arriver à un état ou à un autre, il faut que cette symétrie soit rompue. Les ruptures de symétrie sont fréquentes en physique. Par exemple, si on pose une balle de ping-pong sur le sommet du filet, de manière parfaitement symétrique, elle ne peut pas tomber d'un coté ou de l'autre sans briser cette symétrie. Pourtant il est très rare qu'elle reste en équilibre au sommet du filet : elle finit en général par tomber d'un côté ou de l'autre. Ici encore, un souffle de vent, une petite secousse, ou une imperfection dans la construction de la balle suffit à décider de quel côté elle tombera.

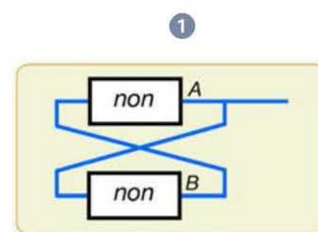
- Si, en revanche, la sortie *A* de la première porte *non* est dans l'état 1, alors l'entrée de la seconde porte *non*, qui est *A* également, est aussi dans l'état 1 ; par conséquent, sa sortie *B* est dans l'état 0, donc l'entrée de la première porte, qui est *B* également, est dans l'état 0, ce qui participe à perpétuer le fait que sa sortie *A* soit dans l'état 1.

Autrement dit, les deux états stables de ce circuit sont :

- $A = 0$ et $B = 1$,
- $A = 1$ et $B = 0$.

En supprimant la sortie *B* et en ne gardant que la sortie *A* (1) on obtient un circuit qui a deux états stables. La sortie *A* vaut 0 dans le premier et 1 dans le second. On peut donc dire que ce circuit *mémorise* la valeur 0 dans le premier cas et la valeur 1 dans le second. Ce circuit est donc un *circuit mémoire*.

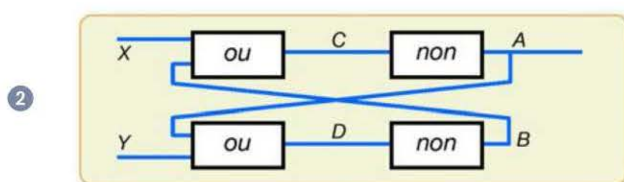
1



Toutefois, ce circuit ayant une sortie, mais pas d'entrée, il n'est pas possible de changer son état et donc la valeur mémorisée.

Pour ce faire, il faut construire un circuit 2, un peu plus complexe, en ajoutant deux portes *ou*.

2



Tant que les entrées *X* et *Y* sont dans l'état 0, tout se passe comme dans le circuit précédent. En effet, si la sortie *A* de la première porte *non* est

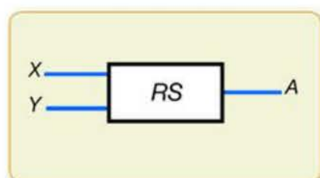
dans l'état 0, alors le point *D* à l'entrée de la seconde est dans l'état 0 également, car 0 *ou* 0 = 0. Et si la sortie *A* est dans l'état 1, le point *D* est dans l'état 1 également, car 1 *ou* 0 = 1. Le point *D* est donc dans le même état que la sortie *A* dans les deux cas. De même, le point *C* à l'entrée de la première porte *non* est dans le même état que *B*. Tout se passe donc comme si les deux portes *ou* n'étaient pas là.

En revanche, si pendant une courte durée on met l'entrée *X* dans l'état 1 tout en laissant l'entrée *Y* dans l'état 0, alors le point *C* passe dans l'état 1 quelle que soit la valeur de *B* car 1 *ou* 0 = 1 et 1 *ou* 1 = 1 ; la sortie *A* passe donc dans l'état 0, le point *D* également et le point *B* passe dans l'état 1. On force donc le circuit à se mettre dans l'état *A* = 0 et *B* = 1, c'est-à-dire à mémoriser la valeur 0. Et quand l'entrée *X* sera revenue à la valeur 0, le circuit restera dans cet état.

Au cours d'étapes qui s'enchaînent très vite, l'état des points *C*, *D*, *A* et *B* devient :

X	Y	C	D	A	B
1	0				
1	0	1			
1	0	1		0	
1	0	1	0	0	
1	0	1	0	0	1

De même, si pendant une courte durée, on met l'entrée *Y* dans l'état 1 tout en laissant l'entrée *X* dans l'état 0, on force le circuit à mémoriser la valeur 1. Ce circuit mémorise donc une valeur 0 ou 1 et, en stimulant l'entrée *X* ou l'entrée *Y*, on peut changer la valeur mémorisée. Ce circuit s'appelle une *bascule RS* (*Reset-Set*) et on peut le représenter comme ci-contre.



À la boîte à outils de circuits commencée au chapitre 13, on peut donc ajouter un premier circuit séquentiel : la bascule *RS*.

On peut aller un peu plus loin et construire un troisième circuit qui mémorise une valeur *V* qu'on lui fournit lorsqu'on stimule l'entrée *S*.

Exercice 14.1 (avec corrigé)

1. Construire un circuit combinatoire à deux entrées *V* et *S* et à deux sorties *X* et *Y* tel que, si l'entrée *S* est dans l'état 0, alors *X* et *Y* sont dans l'état 0 et si *S* est dans l'état 1, alors *X* est dans l'état *non V* et *Y* est dans l'état *V*.
2. Connecter ce circuit combinatoire avec la bascule *RS* ci-avant pour obtenir un circuit qui :
   - quand *S* est dans l'état 0, ne change pas d'état et produit la valeur mémorisée sur la sortie *A*,
   - quand *S* est dans l'état 1 et *V* dans l'état 0, mémorise la valeur 0 et met la sortie *A* dans l'état 0,

- quand S est dans l'état 1 et V dans l'état 1, mémorise la valeur 1 et met la sortie A dans l'état 1.

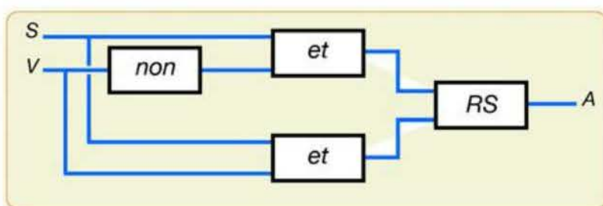
1 *Les tables des fonctions booléennes* X *et* Y *sont les suivantes :*

S	V	X
0	0	0
0	1	0
1	0	1
1	1	0

S	V	Y
0	0	0
0	1	0
1	0	0
1	1	1

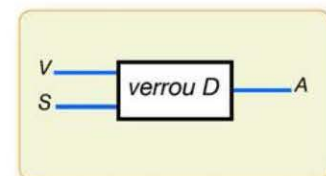
où on reconnaît les fonctions S et non (V) *et* S et V. *Il suffit donc de construire les circuits correspondants.*

2 *On relie la sortie du circuit* S et (non V) *à l'entrée* X *de la bascule* RS *et la sortie du circuit* S et V *à l'entrée* Y *de la bascule* RS.



Ainsi, quand S *est dans l'état 0, le circuit ne change pas d'état et quand* S *est dans l'état 1, il mémorise la valeur* V. *On a donc enfin un véritable circuit mémoire contrôlable : mettre l'entrée* S *dans l'état 1 provoque la mémorisation de la valeur de l'entrée* V. *Cette valeur peut être lue par la suite sur la sortie* A, *jusqu'à ce que l'on provoque la mémorisation d'une nouvelle valeur. Ce circuit s'appelle un* verrou D (D *pour* Donnée ou Data*).*

Verrou D





Exercice 14.2

Quand on utilise plusieurs composants identiques dans un même circuit, on peut différencier leurs entrées et sorties en leur donnant un numéro : ainsi, si l'on a plusieurs verrous D, on appellera V_1 l'entrée V du premier, A_2 la sortie du deuxième, etc.

On définit un circuit séquentiel appelé *bascule D* en reliant la sortie A_1 d'un premier verrou D avec l'entrée V_2 d'un second verrou D, et en reliant l'entrée S_1 du premier à une porte *non* dont la sortie est raccordée à l'entrée S_2 du second.

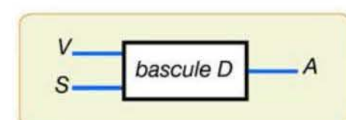
Si l'on considère ce circuit dans son intégralité, ses entrées sont donc l'entrée V_1 du premier verrou D et une entrée S unique qui alimente à la fois S_1 directement et S_2 via la porte *non*. Sa sortie est la sortie A_2 du second verrou D.

Dessiner ce circuit.

Montrer que quand S est dans l'état 0, le second verrou met sa valeur à jour, mais pas le premier, et quand S est dans l'état 1, le premier verrou met sa valeur à jour, mais pas le second.

Ce circuit s'appelle une *bascule D*.

Bascule D



Son intérêt est que la valeur mémorisée n'est mise à jour qu'à l'instant précis où l'entrée S passe de l'état 1 à l'état 0, alors qu'un simple verrou D laisse passer les valeurs de V vers A tant que S reste à 1. Il est ainsi plus facile de commander l'évolution de ce circuit dans le temps.



Exercice 14.3

On utilise un additionneur un bit (voir le chapitre 13) avec deux entrées A et B et deux sorties S et C, qui sont respectivement le chiffre des unités et le chiffre des deuzaines de $A + B$. On relie la sortie S de l'additionneur à l'entrée V_1 d'une première bascule D, la sortie A_1 de cette bascule étant elle-même reliée à l'entrée A de l'additionneur. On relie également la sortie C de l'additionneur à l'entrée V_2 d'une deuxième bascule D. Enfin, on relie ensemble les entrées S_1 et S_2 des deux bascules et on les alimente avec une entrée commune S.

Dessiner ce circuit.

À chaque fois que l'entrée S des bascules passe dans l'état 1 puis revient à l'état 0, les valeurs mémorisées par les bascules D sont mises à jour. Montrer que le nouvel état de la première bascule est le chiffre des unités de la somme de son ancien état et de l'entrée B. Montrer que le nouvel état de la seconde bascule est le chiffre des deuzaines de cette somme.



Exercice 14.4

En utilisant huit copies du circuit construit à l'exercice précédent, construire un compteur 8 bits comportant une entrée I, tel que l'entier 8 bits mémorisé par compteur soit augmenté d'une unité à chaque fois que l'entrée I passe dans l'état 1 puis revient à l'état 0.



Exercice 14.5

Modifier le circuit de l'exercice précédent de manière à ajouter une entrée R telle que le nombre mémorisé par ce compteur soit initialisé à 0 quand l'entrée R est mise dans l'état 1.

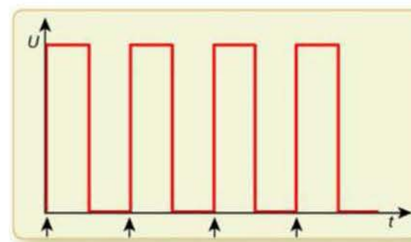
L'horloge

Quand on assemble des circuits séquentiels qui interagissent les uns avec les autres, on obtient un circuit *asynchrone* : l'état de chacun des circuits évolue dans le temps, en fonction de l'état des circuits auxquels il est connecté, mais de manière relativement désordonnée.

Beaucoup d'interactions que nous avons avec les autres sont asynchrones : dans un grand magasin, par exemple, chaque client fait ses courses relativement indépendamment des autres. Et quand plusieurs clients veulent payer leurs courses en même temps, il se forme une file d'attente. De ce fait, chaque client est à peu près sûr qu'il finira par payer et sortir, mais il n'a pas de garantie *a priori* sur le temps que cela prendra. À l'inverse, un petit nombre de nos interactions doivent être synchrones : jouer dans un orchestre demande non seulement de jouer toutes les notes de la partition

dans un certain ordre, mais également de les jouer à un moment donné. Les clients d'un magasin peuvent faire leurs courses chacun à son rythme, mais les musiciens d'un orchestre, les danseurs d'une chorégraphie ou les soldats qui marchent au pas, doivent agir dans une même temporalité. Une manière d'obtenir cette synchronie est de confier à l'un des musiciens, le chef d'orchestre, le rôle de battre la mesure.

En informatique, les machines de grande taille, par exemple les réseaux, sont des machines asynchrones. Chaque utilisateur va à son rythme et le réseau finit par répondre à tout le monde, mais sans garantie du temps que cela prendra. En revanche, les machines de petite taille, telles que les processeurs, sont des machines synchrones. C'est pour cela qu'il y a dans les ordinateurs un circuit, *l'horloge*, dont le rôle est de battre la mesure pour les autres circuits. Une horloge est simplement un circuit qui émet sur sa sortie un signal périodique, par exemple le signal suivant.



Chaque flèche sur la figure marque le début d'un cycle. Avec ce type d'horloge, la sortie est à 1 pendant la première moitié du cycle et à 0 pendant la seconde.

Chacun des circuits, en particulier les circuits mémoires, se synchronise sur un signal d'horloge. Par exemple, si on connecte la sortie de l'horloge sur l'entrée *S* d'une bascule *D*, on obtient un circuit qui enregistre la valeur de l'entrée *V* à chaque cycle.



Exercice 14.6

Construire un circuit comportant un additionneur 8 bits, à 16 entrées et 9 sorties, dont chaque entrée est reliée à la sortie *A* d'un verrou *D* différent, chaque sortie est reliée à l'entrée *V* d'un verrou *D* différent, et dont toutes les entrées *S* de ces 25 verrous sont reliées à une horloge unique.

Établir un lien entre la fréquence de l'horloge et le temps de propagation des signaux électriques portant les retenues des 8 additionneurs un bit composant l'additionneur 8 bits.

Estimer le temps de propagation maximal par porte booléenne, supposé constant et uniforme, compatible avec une fréquence de 1 GHz.



Exercice 14.7

Lorsque l'on relie l'entrée *S* d'une bascule *D* à une horloge, montrer que la sortie *A* se comporte comme l'entrée *V*, mais avec un cycle d'horloge de retard.

Fréquence d'horloge

La *fréquence d'horloge*, qui est souvent indiquée dans les caractéristiques techniques des ordinateurs, est le nombre de cycles par seconde. Par exemple, quand la fréquence d'horloge est 1 GHz, la période de l'horloge, c'est-à-dire la longueur d'un cycle, est de 1 nanoseconde, un milliardième de seconde.

ALLER PLUS LOIN La réversibilité

Le fonctionnement d'un circuit séquentiel est dynamique : il se décrit dans le temps. Comme à chaque fois que l'on observe un phénomène dynamique, on peut s'interroger sur sa réversibilité.

Par exemple, si on connaît la position x et la vitesse v, à un instant t, d'un point en mouvement rectiligne uniforme, on peut prédire sa position et sa vitesse à un instant t' du futur : sa position sera $x' = x + v\,(t' - t)$ et sa vitesse sera $v' = v$. Réciproquement, si on connaît sa position x' et la vitesse v' à l'instant t', on peut aussi « rétro-prédire » sa position et sa vitesse à un instant t du passé : sa position était $x = x' - v'\,(t' - t)$ et sa vitesse était $v = v'$. La variation de position et de vitesse d'un point en mouvement rectiligne uniforme est un phénomène dit réversible.

En revanche, quand on mélange un litre de gaz à la température T_1 avec un litre de gaz à la température T_2, on obtient deux litres de gaz à la température $T' = 2\,T_1\,T_2\,/\,(T_1 + T_2)$. Si on connaît les températures T_1 et T_2, on peut donc prédire la température T'. En revanche, si on connaît la température T', il est impossible de rétro-prédire les températures T_1 et T_2 : deux litres de gaz à 300 K peuvent avoir été obtenus en mélangeant un litre à 290 K et un litre à 310,7 K, mais il peuvent aussi avoir été obtenus en mélangeant un litre à 280 K et un litre à 323 K. Mélanger des gaz de températures différentes est donc un phénomène dit irréversible. Non seulement il est impossible de rejouer le film en arrière et d'obtenir les deux litres de gaz aux températures T_1 et T_2, mais il est de plus impossible de définir ce que ces températures T_1 et T_2 doivent être, puisqu'il y a plusieurs solutions à l'équation $2\,T_1\,T_2\,/\,(T_1 + T_2) = T'$.

Les évolutions des états des circuits vues dans ce chapitre sont des évolutions irréversibles, comme le mélange de deux volumes de gaz de températures différentes. Si l'entrée V d'un circuit mémoire est à 0, au moment où l'on met son entrée S à 1, ce circuit enregistre la valeur 0. La valeur précédemment mémorisée est irrémédiablement détruite. Il est donc impossible de rejouer le film à l'envers pour retrouver l'état initial du circuit, car les deux états initiaux possibles mènent au même état final.

La physique statistique et la théorie de l'information permettent de mesurer le degré d'irréversibilité de ces deux phénomènes : la croissance de l'entropie, ou la perte d'information, lors de la destruction d'un bit d'information est de $9{,}5\ 10^{-24}$ J/K et celle lors du mélange d'un litre d'un gaz parfait monoatomique à 10^5 Pa et 290 K avec un litre de ce même gaz à cette même pression et 310,7 K est de $9{,}9\ 10^{-4}$ J/K, soit en comptant cette perte d'information, non en J/K, mais en bits, 10^{20} bits.

ALLER PLUS LOIN Transformation d'énergie et production de chaleur

Un processeur consomme de l'électricité et la transforme essentiellement en chaleur. L'électricité consommée et la chaleur produite sont des facteurs physiques limitant les performances des processeurs.

Le coût énergétique d'un calcul a beaucoup diminué avec le temps puisqu'on est passé d'une production de calculs d'environ 400 calculs par kWh pour l'ENIAC, en 1946, à environ 1 million de milliards de calculs par kWh pour un processeur actuel. Dans cette estimation, « un calcul » est par exemple l'addition de deux nombres entiers. Cependant, comme on fait beaucoup plus de calculs qu'en 1946, la quantité d'électricité consommée pour ce faire a beaucoup augmenté pour arriver, en 2007, à environ 1 % de la production mondiale d'électricité. Cette estimation ne tient compte que de l'énergie transformée pour effectuer les calculs et non de celle transformée pour fabriquer, transporter et recycler les machines utilisées.

Les gros centres de serveurs de calcul ou de données ont des besoins énormes en électricité et cherchent souvent à se rapprocher géographiquement des centrales électriques. Ils requièrent de même des systèmes de climatisation de plus en plus sophistiqués, par exemple des systèmes de refroidissement qui utilisent de l'eau de mer.

Ai-je bien compris ?

- Quelle est la différence entre un circuit combinatoire et un circuit séquentiel ?
- Qu'est-ce qu'une bascule RS ?
- Quelle est la différence entre un circuit synchrone et un circuit asynchrone ?

L'organisation d'un ordinateur 15

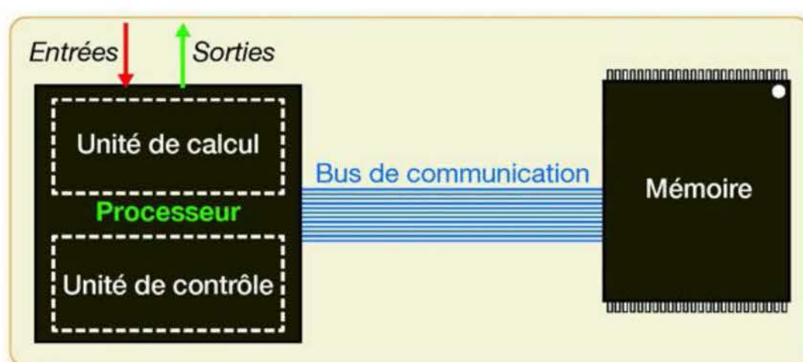


Dans les années 1940, à l'Université de Pennsylvanie, **John von Neumann** (1903-1957) a conçu, avec Presper Eckert et John Mauchly, deux des premiers ordinateurs : l'ENIAC, puis l'EDVAC. Ces ordinateurs étaient organisés selon l'architecture de von Neumann, utilisée dans la quasi-totalité des ordinateurs conçus depuis : séparation du processeur et de la mémoire, reliés par un bus de communication. L'ENIAC pesait vingt-sept tonnes.

Pour fabriquer un ordinateur,
il suffit d'un fer à souder (ou presque...).

Dans ce chapitre, nous voyons de quoi sont faits les ordinateurs à une échelle plus proche de la nôtre et comment architecture et langages sont liés. Nous décrivons la manière dont sont assemblés le processeur de calcul, l'organisation de la mémoire et les bus permettant la circulation des données. Nous voyons comment programmer le processeur au moyen d'un langage machine simple et expliquons comment dérouler une séquence d'instructions. Nous adjoignons enfin les périphériques pour obtenir un ordinateur.

Après avoir décrit le fonctionnement d'un ordinateur à l'échelle du transistor puis de la porte booléenne, nous abordons, dans ce chapitre, une troisième échelle de description, qui est celle qui va nous permettre de véritablement en comprendre les principes d'organisation. Un ordinateur est principalement composé de deux grands circuits : le *processeur* et la *mémoire*. Ces deux circuits sont reliés entre eux par des fils qui constituent un ou plusieurs *bus de communication*, parmi lesquels un *bus de données* et un *bus d'adresses*. Le processeur est composé de deux unités. L'*unité de contrôle* lit en mémoire un programme et donne à l'*unité de calcul* la séquence des instructions à effectuer. Le processeur dispose par ailleurs de *bus d'entrées et de sorties* permettant d'accéder aux autres parties de l'ordinateur, que l'on nomme les *périphériques*. Cette organisation générale, l'*architecture de von Neumann*, est étonnamment stable depuis les années quarante.



> ALLER PLUS LOIN
> **Taille de la mémoire**
>
> En général, on indique la taille de la mémoire en précisant le nombre d'octets, c'est-à-dire de mots de huit bits, qui peuvent être mémorisés. Ainsi, une mémoire de 4 gigaoctets (binaires), contient $4 \times 2^{30} \times 8 = 34\,359\,738\,368$ circuits mémoires un bit. Si la mémoire est organisée en mots de soixante-quatre bits, ces circuits sont répartis en 536 870 912 cases permettant de mémoriser un mot chacune.

> ALLER PLUS LOIN
> **Les processeurs 32 et 64 bits**
>
> Lorsque l'on parle de *processeurs 32 bits* ou *64 bits*, on fait référence à la taille de ces registres.

La mémoire est composée de plusieurs milliards de circuits mémoires un bit. Ces circuits sont organisés en agrégats de huit, seize, trente-deux, soixante-quatre bits, et parfois davantage, que l'on appelle des *cases mémoires* et qui peuvent donc mémoriser des mots de huit, seize, trente-deux, soixante-quatre bits, etc. Le nombre de ces cases définit la taille de la mémoire de l'ordinateur. Comme il faut distinguer ces cases les unes des autres, on donne à chacune un numéro : son *adresse*. La mémoire contient les *données* sur lesquelles on calcule et le *programme* qui décrit le calcul effectué, donné sous la forme d'une séquence d'*instructions*.

Le processeur, de son côté, n'a qu'un très petit nombre de cases mémoires, que l'on appelle des *registres*. On peut imaginer, par exemple, qu'il ne contient que deux registres, appelés *A* et *B*. Les registres peuvent contenir des données, mais aussi des adresses de cases mémoires.

Trois instructions

Pour échanger des données avec la mémoire, le processeur utilise deux procédés qui permettent l'un de transférer l'état d'un registre dans une case mémoire et l'autre de transférer l'état d'une case mémoire dans un registre.

Pour transférer le contenu du registre A dans la case mémoire d'adresse n, le processeur met les différents fils qui composent le bus d'adresses dans un état qui correspond à l'expression en base deux du nombre n et il met les différents fils qui composent le bus de données dans un état qui correspond au contenu du registre. Au signal d'horloge, chaque case de la mémoire compare son propre numéro au numéro arrivé sur le bus d'adresse ; seule la case numéro n se reconnaît et elle met alors ses différentes entrées S (voir le chapitre 14) dans l'état 1, de manière à enregistrer le mot arrivant sur le bus de données. Le procédé symétrique permet au processeur de récupérer une valeur précédemment enregistrée : les informations circulent toujours du processeur vers la mémoire sur le bus d'adresses, mais elles circulent dans l'autre sens sur le bus de données : c'est la case n qui connecte sa sortie au bus de données et c'est le registre qui met ses entrées S à 1 de manière à enregistrer le mot qui arrive sur le bus de données.

Ces deux opérations s'appellent le stockage (`STA`) et le chargement (`LDA`) du contenu d'une case mémoire dans le registre A (ST pour *STore*, LD pour *LoaD*). Il y a bien entendu des opérations similaires pour le registre B (`STB` et `LDB`).

Une autre opération que peut exécuter le processeur est l'addition des contenus des registres A et B. Le résultat de l'opération peut être stocké aussi bien dans le registre A (`ADD A`) que dans le registre B (`ADD B`). De même, `DEC A` *décrémente* la valeur contenue dans le registre A, c'est-à-dire soustrait 1 à la valeur contenue dans le registre A et stocke la valeur ainsi obtenue dans le registre A et `DEC B` réalise le même calcul sur la valeur contenue dans le registre B.

```
LDA 7
LDB 8
ADD A
LDB 9
ADD A
LDB 10
ADD A
STA 11
```

Si, par exemple, le processeur effectue successivement les opérations ci-contre et si, dans l'état initial, la case 7 de la mémoire contient le nombre 42, la case 8 le nombre 68, la case 9 le nombre 47 et la case 10 le nombre 33, l'exécution des huit opérations a comme effet de :

- `LDA 7` charger le contenu de la case 7, soit 42, dans le registre A,
- `LDB 8` charger le contenu de la case 8, soit 68, dans le registre B,
- `ADD A` additionner les contenus des registres A et B et mettre le résultat, 110, dans le registre A,
- `LDB 9` charger le contenu de la case 9, soit 47, dans le registre B,

- `ADD A` additionner les contenus des registres *A* et *B* et mettre le résultat, 157, dans le registre *A*,
- `LDB 10` charger le contenu de la case 10, soit 33, dans le registre *B*,
- `ADD A` additionner les contenus des registres *A* et *B* et mettre le résultat, 190, dans le registre *A*,
- `STA 11` stocker le contenu du registre *A*, soit 190, dans la case 11.

Au bout du compte, cette séquence d'opérations additionne les quatre nombres stockés dans les cases 7, 8, 9 et 10 de la mémoire et stocke le résultat dans la case 11.

Le langage machine

Un ordinateur doit être capable d'exécuter un programme. Il faut donc un moyen d'indiquer au processeur la séquence des instructions qu'il doit exécuter ; par exemple, la séquence `LDA 7`, `LDB 8`, `ADD A`, `LDB 9`, `ADD A`, `LDB 10`, `ADD A`, `STA 11`. Dans les premières machines, des cartes perforées ou un ruban perforé situé à l'extérieur de la machine indiquaient les opérations à effectuer, comme les cartes d'un orgue de Barbarie indiquent les notes à jouer l'une après l'autre. Puis cette idée a été abandonnée au profit d'une autre : celle d'enregistrer le programme dans la mémoire avec les données. Ainsi, on peut exprimer le programme précédent en binaire en décidant par exemple que *A* s'écrit 0, *B* s'écrit 1 et les instructions `LDA`, `LDB`, `STA`, `STB`, `ADD` et `DEC` s'écrivent respectivement 0, 1, 2, 3, 4 et 5. Le programme de notre exemple s'écrit alors 0, 7, 1, 8, 4, 0, 1, 9, 4, 0, 1, 10, 4, 0, 2, 11, ce qui commence à devenir assez difficile à lire, même s'il est facile de passer d'une représentation à l'autre. On peut ensuite stocker ce programme dans la mémoire, en commençant, par exemple, à la case 100 :

adresse	..	..	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	..	..
valeur	..	..	0	7	1	8	4	0	1	9	4	0	1	10	4	0	2	11	..	..

Il suffit maintenant d'ajouter au processeur un nouveau registre qui débute à 100, le *compteur de programme* ou *PC (program counter)*, et à chaque étape, le processeur :

- charge le contenu des cases mémoires d'adresses *PC* et *PC* + 1,
- décode le premier de ces nombres en une instruction (0 devient `LDA`, 1 `LDB`, etc.),

- exécute l'instruction en question,
- et ajoute 2 au registre *PC*.

Enregistrer les programmes en mémoire permet de faire très simplement des boucles et des tests. On ajoute aux instructions précédentes une instruction `JMP` (*jump*) telle que `JMP n` charge simplement le nombre *n*, ou plutôt le nombre *n* - 2 qui sera augmenté de 2 immédiatement après l'exécution du `JMP`, dans le registre *PC* pour détourner le programme de sa route et le forcer à continuer son exécution à l'adresse *n*. De même, l'instruction `JMPZ` (*jump if zero*), qui effectue un saut si le contenu du registre *A* est 0, permet de faire des tests. On ajoute enfin l'instruction `END`, qui termine le programme. En langage machine, on suppose que `JMP`, `JMPZ` et `END` s'écrivent respectivement 6, 7 et 8 et que `END` prend 0 comme argument puisqu'il en faut un. Mais cet argument n'est pas utilisé.

Pour construire une boucle ou un test avec ces nouvelles instructions, il faut tout d'abord trouver une façon de traduire la condition du test ou la condition d'arrêt de la boucle par un test d'égalité à zéro. Par exemple, pour effectuer un test comme `x == 2`, on peut placer la valeur de `x` dans le registre *A*, exécuter deux fois l'instruction `DEC A` et enfin tester si le registre *A* contient 0. Ensuite, on écrit les séquences d'instructions qui correspondent aux différentes branches du test ou au corps de la boucle, et on utilise `JMPZ` et `JMP` pour diriger l'exécution du programme dans l'une ou l'autre de ces séquences. Par exemple, un programme qui lit une valeur `x` dans la case mémoire d'adresse 11, puis recopie la case mémoire d'adresse 12 dans la case mémoire d'adresse 20 si `x` vaut 2, ou la case mémoire d'adresse 13 dans la case mémoire d'adresse 30 dans le cas contraire, peut s'écrire :

100	0	LDA 11
101	11	
102	5	DEC A
103	0	
104	5	DEC A
105	0	
106	7	JMPZ 114
107	114	
108	1	LDB 13
109	13	



110	3	STB 30
111	30	
112	6	JMP 118
113	118	
114	1	LDB 12
115	12	
116	3	STB 20
117	20	
118	8	END
119	0	

ALLER PLUS LOIN

Pourquoi séparer A et B pour ADD et DEC mais pas pour LD et ST ?

Dans le langage machine que nous venons d'inventer, pour que le compteur de programme fonctionne correctement, chaque instruction utilise exactement deux cases mémoire : une pour son nom et une pour son argument. L'argument des instructions ADD et DEC est le nom d'un registre. En revanche, comme les instructions de chargement ont déjà un argument (l'adresse mémoire où aller chercher la donnée à charger), elle ne peuvent pas en avoir un second pour indiquer le registre utilisé. C'est pour cela que l'on a deux instructions LDA et LDB et de même deux instructions STA et STB.

ALLER PLUS LOIN

Calcul sur des structures de données plus complexes

Les instructions que nous venons de présenter ne permettent d'accéder qu'à un nombre limité de cases mémoire, dont les adresses sont les constantes entières qui servent d'arguments aux instructions LDA, STA, LDB et STB. Le calcul sur des structures de données plus complexes, comme des tableaux, nécessite d'autres instructions pour accéder à une case mémoire dont l'adresse est elle-même une donnée, en particulier le résultat d'un calcul. Les processeurs ont bien d'autres instructions encore : des opérations booléennes, des opérations sur les nombres entiers, des opérations sur les nombres flottants, etc.

SAVOIR-FAIRE Savoir dérouler l'exécution d'une séquence d'instructions

Le principe est de suivre, instruction par instruction, l'évolution du programme en observant les effets sur les valeurs contenues dans les registres, y compris le compteur de programme *PC*, et les valeurs contenues dans la mémoire, un peu comme on le ferait pour l'état de l'exécution d'un programme écrit en Java.

Exercice 15.1 (avec corrigé)

Écrire une séquence d'instructions qui multiplie par 5 le nombre contenu dans la case mémoire d'adresse 10 et stocke le résultat dans la case mémoire d'adresse 11.

Pour multiplier par 5, on fait 4 additions, en accumulant le résultat dans le registre A. On note x le nombre rangé à l'adresse 10.

	A	*B*	
`LDA 10`	*x*		*Charger dans A le nombre rangé à l'adresse mémoire 10*
`LDB 10`	*x*	*x*	*Charger dans B le nombre rangé à l'adresse mémoire 10*
`ADD A`	*x+x*	*x*	*Additionner A et B, résultat dans A*
`ADD A`	*x+x+x*	*x*	*Additionner A et B, résultat dans A*
`ADD A`	*x+x+x+x*	*x*	*Additionner A et B, résultat dans A*
`ADD A`	*x+x+x+x+x*	*x*	*Additionner A et B, résultat dans A*
`STA 11`	*x+x+x+x+x*	*x*	*Stocker le nombre contenu dans A à l'adresse mémoire 11*

La séquence LDA 10, LDB 10, ADD A, ADD A, ADD A, ADD A, STA 11, s'écrit en langage machine 0, 10, 1, 10, 4, 0, 4, 0, 4, 0, 4, 0, 2, 11. On la stocke par exemple à partir de l'adresse 100 de la mémoire.

État de la mémoire avant l'exécution du programme :

adresse	..	10	11	..	..	100	101	102	103	104	105	106	107	108	109	110	111	112	113	..
valeur	..	x		..	..	0	10	1	10	4	0	4	0	4	0	4	0	2	11	..

État de la mémoire après l'exécution du programme :

adresse	..	10	11	..	..	100	101	102	103	104	105	106	107	108	109	110	111	112	113	..
valeur	..	x	5x	..	..	0	10	1	10	4	0	4	0	4	0	4	0	2	11	..

Exercice 15.2

Expliquer ce que fait le programme suivant, écrit en langage machine, en supposant que le nombre *x* contenu dans la case mémoire d'adresse 10 est strictement positif.

adresse	..	10	11	..	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	..
valeur	..	x	y		0	10	1	10	7	112	5	0	4	1	6	104	3	11	8	0	..



Exercice 15.3

Écrire un programme qui lit deux valeurs *x* et *y* contenues respectivement dans les cases mémoires 11 et 12, calcule la différence *y* - *x* et stocke le résultat à l'adresse 13. On suppose que ces deux valeurs sont des nombres entiers positifs.

Compléter ce programme pour qu'il stocke la valeur 0 à l'adresse 15 si *x* est égal à *y*, ou la valeur *x* sinon.



Exercice 15.4

Écrire un programme qui multiplie la valeur contenue à la case mémoire 12 par celle contenue dans la case mémoire 13 et stocke le résultat à l'adresse 14. On suppose que ces valeurs sont des nombres entiers positifs.

Quel problème l'écriture de ce programme pose-t-elle ? Quelle modification du processeur permettrait de contourner ce problème et donc de simplifier le programme ?

La compilation

Les premiers programmeurs écrivaient des programmes en langage machine qui ressemblaient à `LDA 7`, `LDB 8`, `ADD A`, `LDB 9`, `ADD A`, `LDB 10`, `ADD A`, `STA 11`, ou plus exactement à 0, 7, 1, 8, 4, 0, 1, 9, 4, 0, 1, 10, 4, 0, 2, 11, ce qui était très fastidieux et offrait de nombreuses possibilités de se tromper. On a alors cherché à concevoir des langages dans lesquels ce programme pouvait s'écrire :

```
e = a + b + c + d;
```

ce qui a permis de développer des programmes de manière plus rapide et plus sûre. Néanmoins, aujourd'hui encore, les ordinateurs ne comprennent que les programmes de la forme 0, 7, 1, 8, 4, 0, 1, 9, 4, 0, 1, 10, 4, 0, 2, 11. C'est pourquoi quand on écrit :

```
e = a + b + c + d;
```

on doit ensuite utiliser un programme qui transforme les programmes écrits en langage évolué en des programmes écrits en langage machine : un *compilateur*. Un programme existe donc toujours sous deux formes : le *code source* écrit en Java, C, etc. et le *code compilé* écrit en langage machine.

Le compilateur est un traducteur d'un langage évolué vers un langage machine. Une manière simple de compiler un programme est de traduire les instructions une à une. Cependant, comme pour les circuits vus au chapitre 13, le compilateur peut construire des programmes en langage machine plus efficaces en se plaçant aussi à l'échelle du langage machine. Par exemple, le temps d'accès à une donnée en mémoire peut être une centaine de fois supérieur au temps d'accès à un registre. Conserver une

valeur dans un registre entre deux instructions au lieu de la stocker pour la recharger ensuite peut donc faire gagner beaucoup de temps de calcul.

Les périphériques

Outre un processeur, une mémoire, une horloge et des bus reliant le processeur à la mémoire, un ordinateur est également constitué de *périphériques* : écrans, claviers, souris, disques, haut-parleurs, imprimantes, scanners, cartes réseau, clés de mémoire flash, etc. qui permettent au processeur d'échanger des informations avec l'extérieur : des êtres humains, à travers par exemple l'écran et le clavier, d'autres ordinateurs, à travers la carte réseau, et des outils de stockage, par exemple des disques. On peut grossièrement classer les périphériques en *périphériques d'entrée*, qui permettent au processeur de recevoir des informations de l'extérieur, et *périphériques de sortie*, qui lui permettent d'émettre des informations vers l'extérieur. Toutefois, beaucoup de périphériques sont à la fois des périphériques d'entrée et de sortie. Ainsi, un écran est a priori un périphérique de sortie, mais un écran tactile est aussi un périphérique d'entrée.

Pour échanger des informations avec les périphériques, le processeur procède d'une manière très similaire à celle qu'il utilise pour échanger des informations avec la mémoire. Par exemple, on peut donner une adresse à chaque pixel d'un écran noir et blanc, exactement comme on donne une adresse à chaque case de la mémoire ; stocker une valeur comprise entre 0 et 255 à cette adresse, avec l'instruction `STA` par exemple, aura pour effet d'allumer ce pixel à l'écran avec le niveau de gris correspondant. De même l'instruction `LDA` permet de recevoir des informations d'un périphérique de sortie, par exemple la position de la souris. Selon les processeurs, ce sont les instructions `STA` et `LDA` elles-mêmes qui sont utilisées, ou des instructions voisines, spécialisées pour la communication avec les périphériques.

Le système d'exploitation

La description des ordinateurs que l'on a donnée dans ce chapitre diffère quelque peu de l'expérience pratique qu'ont tous ceux qui ont déjà utilisé un ordinateur. Tout d'abord, on s'aperçoit que si on bouge la souris, le curseur de souris bouge sur l'écran, il semble donc y avoir un programme qui interroge en permanence la souris pour connaître sa position et dessine un cur-

seur de souris à l'endroit correspondant de l'écran. De même, il est possible d'utiliser simultanément plusieurs programmes, alors que dans la description que nous avons donnée, le processeur n'en exécute qu'un seul à la fois.

Cela est dû au fait que dès qu'on allume un ordinateur, un programme spécial est lancé : *le système d'exploitation*. Ce programme, souvent gigantesque, a plusieurs fonctions :

- Il permet l'exécution simultanée de plusieurs programmes, selon le système du *temps partagé* : le système d'exploitation exécute chacun des programmes à tour de rôle et pendant une courte durée, garantissant à chacun que ses données ne seront pas modifiées par les autres – ainsi, même si un programme croit utiliser les cases mémoire 1, 2 et 3, il se peut qu'il utilise en fait les cases 4097, 4098 et 4099, car le système d'exploitation a réservé les véritables cases 1, 2 et 3 pour un autre programme.
- Il gère les périphériques ; ainsi, pour imprimer un caractère sur l'écran, il n'est pas nécessaire d'allumer chaque pixel l'un après l'autre, mais on peut demander au système d'exploitation d'afficher un « A » et celui-ci traduira cette instruction en une suite d'instructions qui afficheront un « A » pixel par pixel. La partie du système d'exploitation qui gère un périphérique s'appelle un *pilote*.
- En particulier, il gère le disque, son découpage en fichiers, l'attribution d'un nom à chaque fichier et leur organisation arborescente (voir le chapitre 11).
- Il gère aussi l'écran, c'est-à-dire son découpage en fenêtres, l'ouverture et la fermeture des fenêtres.
- Dans certains systèmes utilisés par plusieurs personnes, il gère l'authentification de chaque utilisateur et les droits, en particulier de lecture et d'écriture des fichiers, associés à chacun d'eux.

ALLER PLUS LOIN

Plusieurs systèmes d'exploitation

Il existe plusieurs systèmes d'exploitation : Unix, Linux ou GNU/Linux, Windows, Mac OS, etc. Toutefois, le développement d'un système d'exploitation est une tâche si coûteuse en temps, qu'il n'existe guère plus d'une dizaine de systèmes d'exploitation réellement utilisés.

ALLER PLUS LOIN **Les ordinateurs parallèles**

Une manière de fabriquer des ordinateurs plus rapides est de mettre dans un ordinateur plusieurs processeurs qui calculent *en parallèle*, c'est-à-dire en même temps. Les ordinateurs qui ont plusieurs processeurs parallèles sont appelés *ordinateurs parallèles* ou *multicœurs*. Certains algorithmes sont très faciles à paralléliser : par exemple pour augmenter la luminosité d'une image en niveaux de gris, il faut ajouter une constante à chaque pixel. Chaque pixel peut être traité indépendamment des autres et utiliser deux processeurs au lieu d'un divise le temps de calcul par deux. Toutefois, d'autres algorithmes sont plus difficiles à paralléliser.

La programmation des processeurs parallèles est plus difficile car, en plus d'écrire des instructions, il faut prévoir sur quel processeur elles vont s'exécuter.

Les processeurs peuvent se partager une mémoire ou plusieurs. Si deux processeurs partagent la même mémoire et doivent se communiquer des données, il faut s'assurer que le premier ait bien fini de les calculer et de les stocker en mémoire avant que le second ne les lise. On dit que les deux processeurs doivent se *synchroniser*. Ils peuvent aussi avoir des mémoires différentes et communiquer par un bus. Les processeurs peuvent fonctionner sur la même horloge ; on dit alors qu'ils sont *synchrones*. Ou bien chaque processeur peut avoir sa propre horloge ; on dit qu'ils sont *asynchrones*. Le parallélisme existe aussi à l'intérieur des processeurs, entre les instructions du langage machine. Cette forme de parallélisme s'appelle le *parallélisme d'instructions*.

ALLER PLUS LOIN La hiérarchie mémoire

Il y a une évidente ressemblance entre les notions de case de la mémoire d'un ordinateur et de boîte associée à une variable dans un état de l'exécution d'un programme. Toutefois, ces deux notions ne sont pas absolument identiques. Un programmeur peut faire comme si une valeur stockée dans une boîte de nom x était stockée au même endroit de la mémoire d'un bout à l'autre de l'exécution du programme. Cependant, cette valeur peut en réalité changer de place. Elle peut être en mémoire, mais comme le système d'exploitation gère plusieurs programmes en temps partagé, il peut très bien attribuer à la boîte de nom x d'abord une case de la mémoire, puis une autre plus tard. Ce que le système d'exploitation garantit est que, quoi qu'il se passe en réalité, tout se passera de manière transparente pour le programmeur, qui peut donc faire comme si le contenu de cette boîte était toujours stocké dans la même case.
Si on utilise des processeurs parallèles pour faire un calcul, la valeur de la boîte x peut être dans la mémoire d'un processeur ou d'un autre, voire dans plusieurs mémoires en même temps. On a vu aussi que la valeur peut être dans un registre au moment où elle est utilisée pour un calcul ou bien elle même calculée si elle est le résultat d'un autre calcul.
La réalité est encore un peu plus complexe car les processeurs disposent de *mémoires caches* ou *antémémoires*. Ce sont les mémoires *L1* et *L2* indiquées dans les caractéristiques techniques des processeurs. Ces mémoires sont d'accès plus rapides que la mémoire ordinaire, mais beaucoup moins grandes. Généralement, l'accès à une mémoire cache ne prend que quelques cycles d'horloge, mais sa capacité est limitée à quelques kilooctets ou mégaoctets. Pendant le calcul, la valeur de la boîte x peut aussi se trouver dans une de ces mémoires caches. Un mécanisme matériel gère automatiquement ces deux mémoires et les instructions du langage machine ne permettent même pas de choisir d'utiliser l'une ou l'autre. Toutefois, le principe à retenir est que toute donnée récemment utilisée a de fortes chances d'être encore dans un des caches. Donc éviter de trop éloigner les utilisations d'une même variable dans un programme peut faire gagner beaucoup de temps de calcul.
On a ici un nouvel exemple de la description d'une même réalité à différents niveaux d'abstraction.

ALLER PLUS LOIN La distribution du code source d'un logiciel

Certains auteurs et éditeurs de programmes ne donnent à leurs utilisateurs que le code compilé de leurs programmes. Ils peuvent ainsi garder leurs secrets de fabrication. Les usagers peuvent utiliser ces programmes, mais ne peuvent pas comprendre comment ils fonctionnent.
D'autres, à l'inverse, donnent à leurs utilisateurs à la fois le code compilé et le code source. Cela permet aux utilisateurs qui le souhaitent de comprendre comment le programme est conçu, de l'adapter à leurs propres besoins si le programme n'y répond qu'imparfaitement, ou de vérifier que le programme ne fait rien d'indésirable, par exemple qu'il ne contient pas d'espion qui communique à son insu des informations sur l'utilisateur à l'auteur du programme. Cela permet aussi aux utilisateurs de contribuer à l'amélioration du programme, en signalant des erreurs, en les corrigeant ou en ajoutant des composants aux programmes.
Les partisans de la distribution du code source des logiciels articulent leurs arguments sur deux plans : un plan éthique et philosophique et un plan scientifique et technique. Certains insistent sur l'impératif moral de distribuer le code source de ses programmes, avec l'objectif de diffuser à tous des connaissances et de leur permettre de se les approprier. D'autres insistent sur le fait que permettre aux utilisateurs de contribuer aux logiciels développés en améliore la qualité. De fait, certains logiciels tels les systèmes d'exploitation GNU/Linux sont aujourd'hui développés par des milliers de contributeurs de par le monde, ce qui serait impossible si le code source n'était pas publié.

ALLER PLUS LOIN **Des programmes auto-modifiants aux virus**

Stocker le programme à exécuter dans la mémoire, avec le reste des données, a permis de prendre conscience de la possibilité pour un programme de se modifier lui-même, par une simple instruction STA, dans la partie de la mémoire consacrée au stockage du programme. Si cette possibilité disparaît dans les langages de programmation évolués, elle est bien présente dans les programmes directement écrits en langage machine.

Cette possibilité de se dupliquer et de se transformer est en particulier utilisée par les *virus informatiques*. Un *virus* est un programme qui est conçu pour se dupliquer et se propager d'ordinateur en ordinateur à l'insu de leurs utilisateurs, par le réseau, ou tout autre support permettant de transmettre de l'information : clé de mémoire flash, CD-ROM, etc. Une fois installé sur un ordinateur, un virus peut espionner ses utilisateurs et communiquer les informations collectées par le réseau. Il peut aussi simplement utiliser la capacité de calcul de l'ordinateur hôte, par exemple pour envoyer des courriers en très grand nombre.

Ai-je bien compris ?

- Quels sont les principaux circuits d'un ordinateur ?
- Qu'est-ce que le langage machine ?
- Qu'est-ce que compiler un programme ?


VOUS VOULEZ VOIR UNE ILLUSION D'OPTIQUE ?
TENEZ VOTRE CLAVIER FACE À VOUS, ET REGARDEZ LA TOUCHE « DÉBUT ».
MAINTENANT, LOUCHEZ UN PEU POUR QUE LE « G » ET LE « H » SE SUPERPOSENT.
MAINTENEZ VOTRE REGARD FOCALISÉ ET LEVEZ LE CLAVIER AU-DESSUS DE VOTRE TÊTE.
ARGH !
MOUHHHAHA !

XKCD

Les réseaux

16

Chapitre avancé

Les ordinateurs parlent aux ordinateurs.

Dans ce chapitre, nous voyons comment les ordinateurs communiquent entre eux, et comment ces communications se composent pour faire fonctionner le réseau Internet. Ces mécanismes de communication de machine à machine s'appellent des protocoles.

Les protocoles de la couche physique connectent les bus des ordinateurs. Les protocoles de la couche lien organisent un réseau local autour d'un serveur et repèrent les ordinateurs par l'adresse MAC de leur carte réseau. Les protocoles de la couche réseau organisent les réseaux locaux de proche en proche et repèrent les ordinateurs par leur adresse IP.

Nous expliquons comment les informations sont acheminées au travers du réseau à l'aide de routeurs.



Vinton Cerf (1943-) et **Robert Kahn** (1938-) ont inventé, au début des années 1970, le protocole de transmission de paquets de données IP (*Internet Protocol*) et le protocole de contrôle de flux de données TCP (*Transmission Control Protocol*). Il s'agit des deux principaux protocoles du réseau Internet. Ils donnent à ce réseau sa fiabilité, sa robustesse en cas de pannes ou de modifications et sa capacité à évoluer. Cela a valu à leurs auteurs le surnom de « pères d'Internet. »

Réseau

Un *réseau* est un ensemble d'ordinateurs et de connexions qui permettent à chaque ordinateur de communiquer avec tous les autres, éventuellement en passant par des intermédiaires.

Nous avons vu qu'un ordinateur pouvait se décrire à différentes échelles : les transistors s'assemblent en portes booléennes, qui s'assemblent à leur tour en composants – processeurs, mémoires, etc. – qui s'assemblent à leur tour en ordinateurs. Et nous pouvons continuer, car les ordinateurs s'assemblent à leur tour en réseaux de différentes tailles : des réseaux les plus simples, formés de deux ordinateurs reliés par un câble ou par radio, aux réseaux locaux connectant quelques ordinateurs entre eux – les ordinateurs d'un lycée par exemple –, qui s'assemblent à leur tour pour former le plus grand des réseaux : *Internet*, lequel relie presque tous les ordinateurs du monde.

Les protocoles

Protocole

Un *protocole* est un ensemble de règles qui régissent la transmission d'informations sur un réseau. Il existe de nombreux protocoles, chacun spécialisé dans une tâche bien précise.

Couche

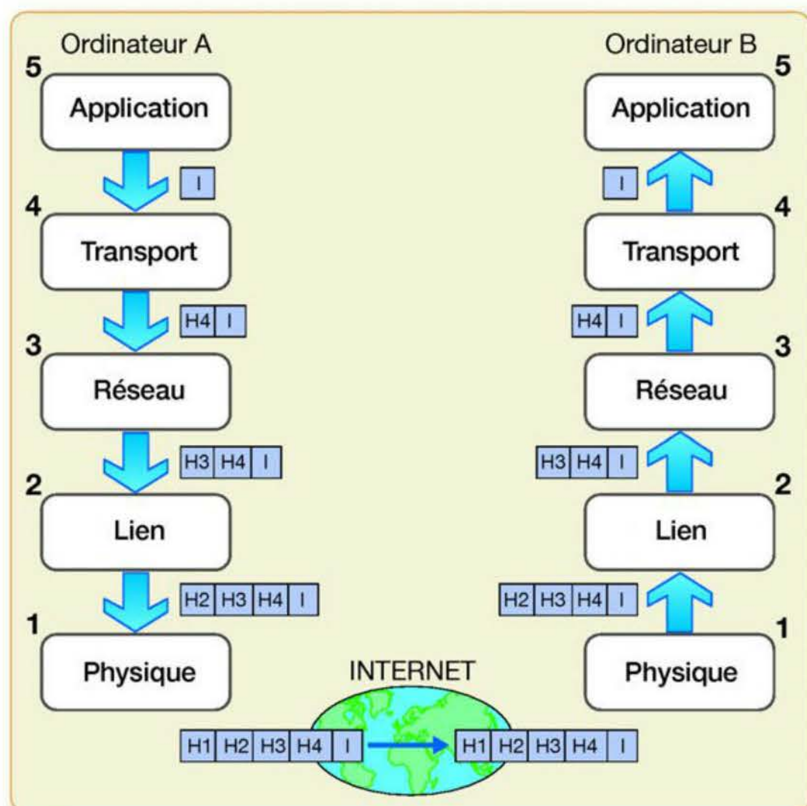
Une *couche* est un ensemble de protocoles qui effectuent des tâches de même niveau. On distingue cinq couches appelées *couche application*, *couche transport*, *couche réseau*, *couche lien* et *couche physique*.

Si un programme P_A, par exemple un logiciel de courrier électronique, exécuté sur un ordinateur A, veut communiquer des informations à un autre programme P_B, exécuté sur un ordinateur B, il sous-traite cette tâche à un programme spécialisé Q_A, exécuté sur l'ordinateur A, qui met en œuvre un *protocole*. Ce programme Q_A dialogue, suivant les spécifications de ce protocole, avec un programme homologue Q_B exécuté sur l'ordinateur B, ce qui permet ainsi la communication entre les programmes P_A et P_B.

En fait, le programme Q_A sous-traite, à son tour, certaines tâches moins sophistiquées à d'autres programmes mettant en œuvre d'autres protocoles, qui sous-traitent, de même, certaines tâches encore plus élémentaires à d'autres protocoles, etc. On peut ainsi classer les protocoles en *couches* hiérarchiques, par le niveau de sophistication des tâches qu'ils exécutent.

Ainsi, les informations envoyées par le programme de courrier électronique sont d'abord confiées à un protocole de la couche application, qui les confie à un protocole de la couche transport, qui les confie à un protocole de la couche réseau, qui les confie à un protocole de la couche lien, qui les confie à un protocole de la couche physique, qui les transmet effectivement vers l'ordinateur B.

Quand on confie une lettre à un facteur, on doit la mettre dans une enveloppe et ajouter sur l'enveloppe des informations supplémentaires : l'adresse du destinataire, sa propre adresse, une preuve de paiement, etc. De même, quand un protocole de la couche k+1 confie des informations à un protocole de la couche k, celui-ci ajoute à ces informations un *en-tête* H_k qui contient des informations, comme l'adresse de l'ordinateur destinataire, utilisées par le protocole de la couche k. On appelle cela l'*encapsulation* des informations. Quand les informations I confiées par la



Système en couches, piles de protocoles. Encapsulation et décapsulation des informations.

couche application à la couche transport arrivent à un protocole de la couche physique, plusieurs en-têtes H_4, H_3, H_2, H_1 leur ont été ajoutés. Ces en-têtes sont supprimés à la réception : la couche k analyse puis supprime H_k avant de passer l'information à la couche $k+1$. On appelle cela la *décapsulation* des informations.

ALLER PLUS LOIN **Les normes**

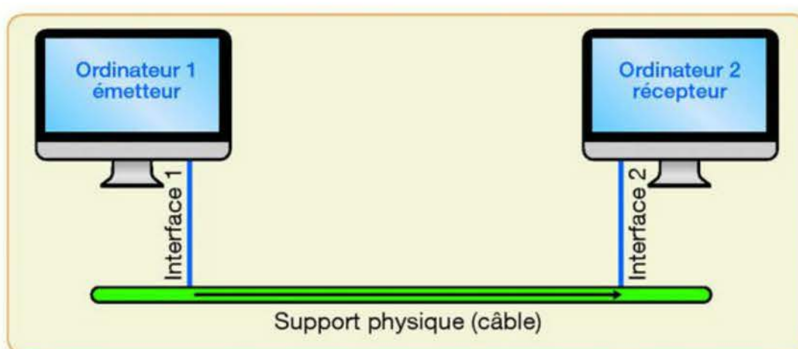
Des *normes* régissent les rôles de chaque couche, leurs interactions et les spécifications de chaque protocole. Ces normes permettent au réseau Internet de fonctionner à l'échelle mondiale et assurent la modularité du système : il est possible de modifier les protocoles à l'œuvre au sein d'une couche, sans modifier les protocoles des autres couches, et le système continue à fonctionner dans son ensemble. Cette modularité est analogue au fait de pouvoir changer un composant matériel d'un ordinateur, par exemple sa carte graphique, sans devoir rien changer d'autre. Cette modularité est essentielle pour permettre au système d'évoluer.

La communication bit par bit : les protocoles de la couche physique

Commençons la présentation de ces protocoles par ceux de la couche physique. Un protocole de la couche physique doit réaliser une tâche extrêmement simple : communiquer des bits entre deux ordinateurs reliés par un câble ou par radio.

Pour relier deux ordinateurs par un câble, et donc réaliser le réseau le plus simple qui soit, on pourrait prolonger le bus de l'un des ordinateurs, afin de le connecter au bus de l'autre. Ainsi, le processeur du premier ordinateur pourrait écrire, avec l'instruction STA, non seulement dans sa propre mémoire, mais aussi dans celle du second. Le processeur du second ordinateur pourrait alors charger, avec l'instruction LDA, la valeur écrite par celui du premier.

La technique utilisée en réalité n'est pas beaucoup plus compliquée : le processeur du premier ordinateur envoie des informations par le bus vers l'un de ses périphériques, *la carte réseau*, qui les transmet par un câble – ou par un autre support physique, par exemple par radio – à la carte réseau du second ordinateur qui les transmet, par le bus, à son processeur.



Transmission point à point

La transmission d'une carte réseau à l'autre met en œuvre un protocole, appelé *protocole physique*. Un exemple de protocole physique est la communication par codes-barres à deux dimensions, comme les pictogrammes *Flashcode* utilisés par les téléphones, où chaque bit est exprimé par un carré noir ou blanc.



Flashcode exprimant le nombre 5412082001000261

Les protocoles physiques qui permettent à deux ordinateurs de communiquer par câble ou par radio ne sont pas très différents. Cependant, au lieu d'utiliser des carrés noirs et blancs, ils représentent les informations par des signaux électromagnétiques, par exemple des variations de longueur d'onde, de phase ou d'intensité d'une onde.

Exercice 16.1

On utilise un lien physique peu fiable : à chaque fois que l'on transmet un 0 ou un 1, la probabilité que ce bit ne soit pas reconnaissable à l'arrivée est 3/10. Pour pallier ce manque de fiabilité, on utilise une forme de redondance (voir le chapitre 12) : quand l'ordinateur émetteur demande à sa carte réseau d'envoyer un 0, cette dernière envoie la suite de bits 0, 1, 0, 0, 1, qui est interprétée par la carte réseau de l'ordinateur récepteur comme un 0. De même, quand l'ordinateur émetteur demande à sa carte réseau d'envoyer un 1, elle envoie la suite de bits 1, 0, 1, 1, 0 qui est interprétée par la carte réseau de l'ordinateur récepteur comme un 1.

1. À partir de combien de bits erronés la suite de cinq bits envoyée n'est-elle plus discernable de l'autre suite ?
2. En déduire la probabilité qu'une suite de cinq bits envoyée ne soit pas reconnaissable à l'arrivée.
3. Quels sont les avantages et inconvénients de cette méthode ?

SUJET D'EXPOSÉ **Protocoles et codages**

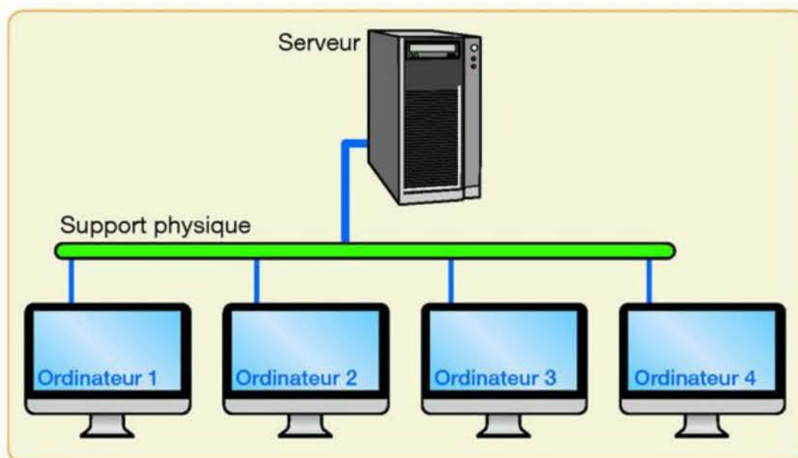
Chercher sur le Web quels sont les protocoles utilisés dans les liaisons RS-232 d'une part, et dans les liaisons USB d'autre part. Quels sont les avantages d'USB par rapport à RS-232 ? Chercher de même ce que sont les codages Manchester d'une part et NRZI d'autre part. Quels sont les avantages du codage Manchester par rapport au codage NRZI ?

Les réseaux locaux : les protocoles de la couche lien

L'étape suivante consiste à construire un réseau local, c'est-à-dire formé de quelques machines connectées, par un protocole physique, à un ordinateur central : un *serveur*. Pour envoyer des informations à un autre ordinateur, chaque ordinateur passe par le serveur.

De même qu'il était nécessaire de distinguer les différentes cases de la mémoire d'un ordinateur en donnant à chacune un nom, son *adresse*, il est nécessaire de distinguer les différents ordinateurs d'un réseau local en donnant à chacun un nom : son *adresse MAC* (*Medium Access Control*). Une adresse MAC est un mot de 48 bits que l'on écrit comme un sextuplet de nombres de deux chiffres en base seize, les seize chiffres s'écrivant 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f : par exemple 10:93:e9:0a:42:ac. Une adresse MAC unique est attribuée à chaque carte réseau au moment de sa fabrication. L'adresse MAC d'une carte réseau, périphérique d'un ordinateur, identifie ce dernier sur le réseau local.

Réseau local



Les protocoles grâce auxquels différents ordinateurs, connectés au même serveur, échangent des informations entre eux sont appelés des *protocoles de la couche lien*. Un protocole simple consiste, pour le serveur, à échanger des informations successivement avec chaque ordinateur du réseau. Ainsi, chaque ordinateur a un créneau de temps pré-établi qui lui est dédié périodiquement pour communiquer avec le serveur, durant lequel le protocole physique épelle bit à bit l'information à transmettre.

Toutefois, les protocoles lien les plus utilisés, par exemple Ethernet et WiFi, utilisent un autre mécanisme qui autorise chaque ordinateur à envoyer des informations au serveur à n'importe quel moment – toujours via un protocole physique. Ce mécanisme évite de réserver un créneau pour un ordinateur qui n'a peut-être rien à communiquer au serveur. Cependant, quand plusieurs ordinateurs envoient des informations en même temps, les messages se brouillent – on dit qu'il y a une *collision entre les messages* – et le serveur n'en reçoit aucun. Pour résoudre ce problème, on utilise une forme de redondance : quand le serveur reçoit les informations envoyées par un ordinateur, il en accuse réception et tant qu'un ordinateur n'a pas reçu d'accusé de réception, il renvoie son message périodiquement. Pour minimiser les risques de nouvelles collisions, ce message est en général renvoyé après un délai de longueur aléatoire. Ce protocole est un exemple d'algorithme qui ne peut fonctionner sans aléa.

Paquet

Un *paquet* est une suite de bits structurés selon un certain format, destinée à être échangée sur le réseau.

Le rôle des protocoles de la couche lien est également de définir le format des messages échangés : de même qu'un fichier PGM (voir le chapitre 9) n'est pas simplement une suite de pixels, mais contient aussi d'autres informations – la largeur et la hauteur de l'image, le nombre de niveaux de gris, etc. – et est structuré selon un certain format standard,

les messages qui circulent sur les réseaux, les *paquets*, ne sont pas simplement formés des bits transmis, mais contiennent des informations additionnelles et sont structurés selon un format standard.

Exercice 16.2

Un pictogramme n'est pas simplement une suite de carrés blancs et noirs, mais il contient aussi des informations qui permettent au téléphone d'en trouver les bords et l'orientation. Quelle est la partie du format de chacun de ces pictogrammes qui joue ce rôle ?



Combien de pictogrammes différents existe-t-il dans chacun de ces formats ? Comment ce nombre se compare-t-il au nombre de pages existant sur le Web ?

SAVOIR-FAIRE Trouver les adresses MAC des cartes réseau d'un ordinateur

Ouvrir une fenêtre *terminal*. Utiliser la commande `ifconfig` sous Linux, ou `ipconfig /all` sous Windows.

Exercice 16.3 (avec corrigé)

Trouver les adresses MAC des cartes réseau de son ordinateur.

Lorsqu'on tape la commande `ifconfig` *ou* `ipconfig /all`*, différentes informations s'affichent, relatives à la connexion de son ordinateur au réseau. Ces informations sont généralement organisées en paragraphes qui correspondent à différentes cartes réseaux : par exemple un paragraphe qui correspond à la carte Ethernet qui gère la connexion par câble et un autre qui correspond à la carte WiFi qui gère la connexion par radio. On reconnaît dans chacun de ces paragraphes une adresse de la forme 10:93:e9:0a:42:ac : un sextuplet de nombres de deux chiffres en base seize. Il s'agit de l'adresse MAC de chacune de ces cartes.*

Exercice 16.4

Noter l'adresse MAC d'une carte réseau de son ordinateur. De quelle connexion réseau s'agit-il précisément ? WiFi ? Ethernet ? Autre ? Éteindre, puis rallumer l'ordinateur. L'adresse a-t-elle changé ?

Le réseau global : les protocoles de la couche réseau

Utiliser un ordinateur central est possible pour un réseau de petite taille, mais cette méthode ne peut pas s'appliquer à un réseau formé de plusieurs milliards d'ordinateurs, comme Internet : d'une part, l'ordinateur central serait vite surchargé, d'autre part s'il tombait en panne, ou s'il était détruit, les communications sur la Terre entière seraient interrompues. C'est pour cela que l'on a inventé d'autres protocoles pour les réseaux de grande taille, appelés *les protocoles réseau*, qui fédèrent les réseaux locaux de proche en proche et utilisent les protocoles lien pour assurer les communications locales. Le plus utilisé des protocoles réseau est le protocole IP (*Internet protocol*).

POUR ALLER PLUS LOIN **D'IPv4 à IPv6**

Une adresse étant un mot de 32 bits, il n'y a que 2^{32} adresses IPv4 possibles, soit un peu plus de quatre milliards : c'est peu quand on sait que le réseau Internet contient déjà trois milliards d'ordinateurs. Cette pénurie en adresses IP, due à l'imprévoyance des pionniers d'Internet, qui cherchaient seulement à connecter quelques dizaines d'ordinateurs sans anticiper le succès de leur réseau, explique que l'on cherche aujourd'hui à les remplacer par des adresses de 128 bits (IPv6).

Avec le protocole IP, chaque machine a une adresse, appelée son *adresse IP*. Contrairement à l'adresse MAC, celle-ci n'est pas associée de manière durable à un ordinateur : quand un ordinateur est remplacé par un autre, le nouveau peut hériter de l'adresse IP de l'ancien. À l'inverse, si un ordinateur est déplacé d'un lieu à un autre, il change d'adresse IP. Les adresses IP classiques (IPv4) sont des mots de 32 bits écrits sous forme d'un quadruplet de nombres compris entre 0 et 255, par exemple 216.239.59.104.

SAVOIR-FAIRE Trouver l'adresse IP attribuée à un ordinateur

Ouvrir une fenêtre *terminal*. Utiliser la commande `ifconfig` sous Linux ou `ipconfig /all` sous Windows.

Exercice 16.5 (avec corrigé)

Trouver les adresses IPv4 attribuées à son ordinateur.

Lorsqu'on tape la commande `ifconfig` *ou* `ipconfig /all`*, différentes informations s'affichent, relatives à la connexion de son ordinateur au réseau. On reconnaît parmi ces informations des adresses de la forme 216.239.59.104 : un quadruplet de nombres compris entre 0 et 255. Il s'agit des adresses IPv4 attribuées à chacune des cartes réseaux de l'ordinateur.*

Exercice 16.6

Noter les adresses IP attribuées aux cartes réseau de son ordinateur. Éteindre, puis rallumer cet ordinateur. Ces adresses ont-elle changé ?

Exercice 16.7

Soit un réseau connectant cinq ordinateurs : *A*, *B*, *C*, *D* et *E*. Est-il possible d'identifier chaque ordinateur de manière unique avec des adresses de 3 bits ? Décrire un tel plan d'adressage : attribuer une adresse à chaque ordinateur. Est-il également possible d'identifier chaque ordinateur de manière unique avec des adresses de 2 bits ?

Exercice 16.8

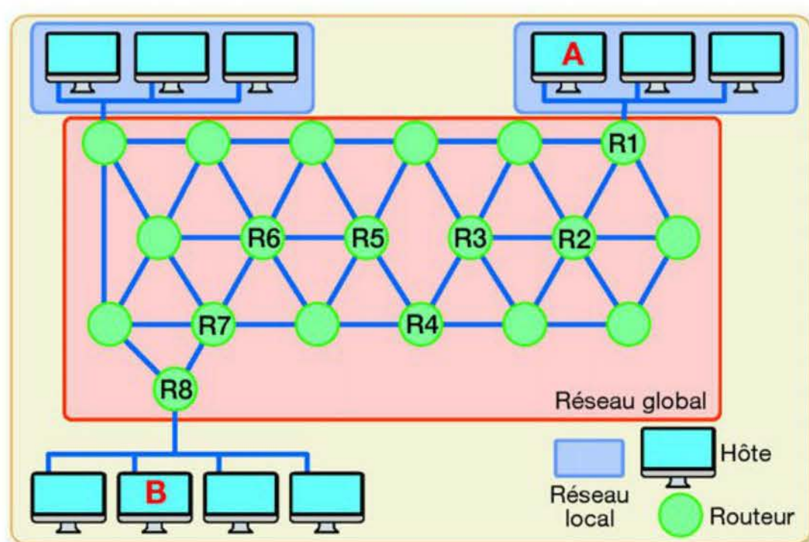
Combien y a-t-il d'adresses IPv4 ? Combien y a-t-il d'adresses IPv6 ? Combien y a-t-il de numéros de téléphone en France ?

Avec les protocoles réseau, la notion de serveur est remplacée par celle de *routeur*. Pour envoyer des informations à une machine dont l'adresse IP est *X*, un ordinateur commence par les envoyer au routeur auquel il est connecté. En fonction de l'adresse *X*, celui-ci l'envoie à un autre routeur, puis à un autre, etc. jusqu'à ce que les informations arrivent à destination. On appelle ce procédé le *routage*.

Routeur

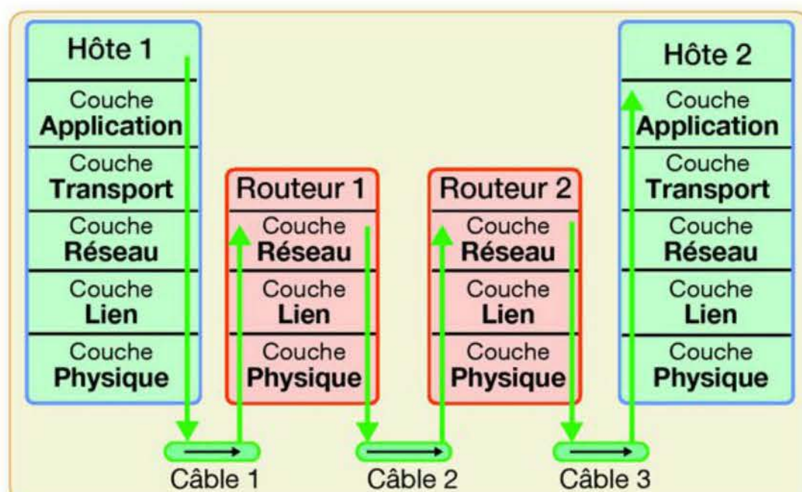
Un *routeur* est un ordinateur dont la seule fonction est d'acheminer des informations sur le réseau. Pour les distinguer des routeurs, on appelle les autres ordinateurs du réseau des *hôtes*.

Les routeurs n'ayant pour fonction que de relayer les paquets en direction de leur destination, ils ne mettent en œuvre que des protocoles des couches physique, lien et réseau, contrairement aux hôtes qui, eux, mettent en œuvre des protocoles de toutes les couches.



Réseau global. Routage de A à B en passant par les routeurs R1, R2, R3, R4, R5, R6, R7 et R8.

La figure suivante illustre le routage d'informations d'un hôte à un autre, en passant par deux routeurs et trois câbles : en chemin, sur chaque machine les traitant, les informations transmises sont successivement encapsulées quand elles passent d'une couche à la couche inférieure, puis décapsulées quand elles passent d'une couche à une couche supérieure.



Les protocoles à l'œuvre lors du routage. Chemin suivi par l'information à travers les couches sur chaque machine.

On peut comparer le routage au système du courrier postal. Si, à Sydney, un Australien expédie une lettre à l'adresse « 2004, route des Lucioles, Valbonne, France », le facteur australien n'apporte pas cette lettre directement à son destinataire, contrairement à ce que faisaient les « facteurs » au XVIIe siècle : il la met dans un avion en direction de Hong-Kong ou de Los Angeles, où un autre facteur la met dans un avion en direction de Francfort ou de Paris, où un autre facteur la met dans un avion en direction de Nice, où un facteur l'apporte finalement en camionnette, au 2004 de la route des Lucioles, à Valbonne.

Un routeur, qui reçoit des informations à envoyer à une adresse IP *X*, doit choisir le routeur suivant, en direction de *X*. De même que le facteur australien sait qu'une lettre pour Valbonne, ou plus généralement pour la France, peut passer par Hong-Kong, le routeur a un répertoire, appelé *table de routage*, qui indique que la première étape d'un chemin vers l'adresse *X* est le routeur dont l'adresse IP est *B*, auquel il est directement connecté. Il envoie donc ces informations vers le routeur *B*, qui consulte à son tour sa propre table de routage, et ainsi de suite jusqu'à arriver à l'ordinateur dont l'adresse est *X*. Comme dans un jeu de piste, on ne connaît pas le chemin à l'avance, mais, à chaque étape, on découvre la suivante.

Pour construire et maintenir à jour leurs tables de routage, les routeurs utilisent un *algorithme de routage*, par exemple *l'algorithme de Bellman-Ford* : outre acheminer des informations d'un point à un autre, chaque routeur envoie périodiquement à chaque routeur auquel il est connecté directement par un protocole lien, la liste des adresses IP vers lesquelles il connaît un chemin, dont bien entendu sa propre adresse, et la longueur

de ce chemin. Grâce à cet algorithme et au fur et à mesure qu'il reçoit des listes, chaque routeur découvre peu à peu l'état du réseau et, ce faisant, les chemins les plus courts pour atteindre les destinations présentes dans le réseau. Ainsi, un routeur qui reçoit du routeur *B* l'information que *B* a reçu du routeur *C* l'information que *C* est directement connecté à l'ordinateur *X*, met à jour sa table de routage en mémorisant qu'un chemin pour accéder à l'ordinateur *X*, en passant par deux intermédiaires, commence par le routeur *B*.

Cette mise à jour dynamique des tables de routage est ce qui donne sa souplesse au réseau Internet. Si par exemple le câble entre les routeurs *B* et *C* est détruit, les tables de routage se mettent graduellement à jour en découvrant d'autres chemins pour acheminer les informations vers la machine *X*. À l'origine, cette idée servait à donner une robustesse au réseau en temps de guerre : tant que deux points restaient connectés par un chemin dans le graphe des routeurs, il leur était possible de communiquer. Par la suite, on s'est rendu compte que cette souplesse était de toutes façons nécessaire dans un réseau de trois milliards d'ordinateurs, où il est inévitable que des ordinateurs soient en permanence ajoutés ou supprimés du réseau.

Si les protocoles réseau sont très souples, ils sont en revanche peu fiables : quand un routeur est saturé par trop d'informations à transmettre, il en détruit tout simplement une partie. En outre, quand des informations ont été envoyées d'un routeur à un autre trop longtemps sans arriver à destination, elles sont également détruites.

SAVOIR-FAIRE Déterminer le chemin suivi par l'information

Dans une fenêtre *terminal*, la commande `ping` suivie d'un nom de domaine affiche l'adresse IP associée à ce nom de domaine. La commande `traceroute` sous Linux ou `tracert` sous Windows, suivie d'une adresse IP, affiche les routeurs d'un chemin menant de son ordinateur à celui dont l'adresse IP est indiquée.

Exercice 16.9 (avec corrigé)

Trouver un chemin entre son ordinateur et l'ordinateur associé au nom de domaine www.google.fr.

Dans une fenêtre terminal*, on tape la commande* `ping www.google.fr` *et une adresse IP s'affiche, par exemple 173.194.34.55. C'est l'adresse d'un ordinateur de Google. On tape ensuite la commande* `traceroute 173.194.34.55` *ou* `tracert 173.194.34.55` *et une liste s'affiche des routeurs qui forment un chemin entre sa machine et celle de Google.*

Savoir-faire Déterminer l'adresse IP du serveur par lequel un ordinateur est connecté à Internet

Dans une fenêtre *terminal*, la commande `netstat -r` sous Linux ou `ipconfig /all` sous Windows affiche différentes informations relatives à la connexion de son ordinateur au réseau, en particulier la *passerelle par défaut* (*default gateway*) qui est le routeur par lequel cet ordinateur est connecté à Internet.

Exercice 16.10 (avec corrigé)

Déterminer l'adresse IP du routeur par lequel son ordinateur est connecté à Internet.

On tape la commande `netstat -r` *ou* `ipconfig /all`*. Dans les informations affichées, on recherche la passerelle par défaut (*default gateway*) indiquée, qui est le routeur. Ce dernier est identifié soit directement par son adresse IP, soit par un nom de la forme* `gw.adalovelace.fr` *; dans ce second cas, on peut trouver l'adresse IP associée à ce nom avec la commande* `ping`*.*

Exercice 16.11

Un algorithme de routage. On attribue à chaque élève une nouvelle adresse mail, par exemple `anonyme1@adalovelace.fr`, `anonyme2@adalovelace.fr`, etc. Chaque élève garde cette adresse secrète et il la marque sur une feuille de papier. Ces feuilles sont mélangées dans un chapeau et chacun tire une adresse en s'assurant qu'il ne tire pas la sienne propre. Chaque élève a uniquement le droit d'envoyer des courriers à l'adresse qu'il a tirée dans le chapeau et de répondre à l'envoyeur de tout courrier qu'il reçoit à son adresse .

Chaque élève envoie un courrier à l'adresse qu'il a tirée, dans lequel il indique son propre nom. Par exemple, Alice enverrait :

```
sujet : Hello
corps du message :
Alice (par -)
```

Chaque élève construit une table de routage qui indique, pour chaque élève dont il a entendu parler, l'adresse de la personne par qui il en a entendu parler pour la première fois. Par exemple, la table de routage d'Alice peut avoir cette forme :

```
Alice (par -)
Djamel (par anonyme2@adalovelace.fr)
Frédérique (par anonyme2@adalovelace.fr)
Hector (par anonyme7@adalovelace.fr)
```

À chaque fois qu'un élève reçoit un message, il le lit, met à jour sa table de routage et y répond en copiant, dans le corps du message l'état de sa table de routage mise à jour. Alice enverrait par exemple :

```
sujet : Hello
corps du message :
Alice (par -)
```

```
Djamel (par anonyme2@adalovelace.fr)
Frédérique (par anonyme2@adalovelace.fr)
Hector (par anonyme7@adalovelace.fr)
```

Au bout d'une dizaine de minutes, on arrête d'envoyer et de répondre aux messages.

On passe alors à la seconde phase de l'exercice : un élève essaie d'envoyer un véritable message à un autre élève. Pour cela, il envoie son message à l'adresse correspondant au nom de son destinataire dans sa table de routage.

Par exemple :

Alice envoie à l'adresse `anonyme2@adalovelace.fr` le message suivant :

```
Sujet : message pour Frédérique
Corps du message : N'oublie pas de me rapporter l'exemplaire de
Madame Bovary que je t'ai prêté il y a six mois. Merci. Alice.
```

Si un élève autre que son destinataire reçoit ce message, il le renvoie à l'adresse correspondant au nom de son destinataire dans sa propre table de routage, et ainsi de suite jusqu'à ce que le message arrive à destination.

À quelle condition un message arrive-t-il bien à son destinataire ?

La régulation du réseau global : les protocoles de la couche transport

Les programmes que l'on utilise tous les jours, par exemple les navigateurs web ou les programmes de gestion du courrier électronique, ne peuvent pas directement utiliser le protocole IP, principalement pour deux raisons : d'une part, le protocole IP ne permet de transférer d'un ordinateur à un autre que des informations de taille limitée : en général, des paquets de 1500 octets au maximum. D'autre part, comme on l'a vu, IP est un protocole peu fiable : dès qu'un serveur est surchargé, il détruit des informations qui n'arrivent donc pas à leur destinataire. On utilise donc un type supplémentaire de protocole : *les protocoles de transport*, dont le plus utilisé est le protocole TCP (*Transmission Control Protocol*). Les protocoles de transport utilisent les protocoles réseau pour acheminer des informations contenues dans des paquets IP, d'un bout à l'autre du réseau, et assurent que tout paquet IP envoyé est arrivé à bon port.

Pour poursuivre la comparaison avec le service postal, si on attache une importance particulière à une lettre, on l'envoie en recommandé et on attend un accusé de réception, qui permet de savoir que sa lettre a bien été reçue. La couche transport s'assure que la communication a bien lieu de bout en bout, comme prévu.

Comme les protocoles de la couche lien, TCP utilise une forme de redondance pour fiabiliser les communications. Il utilise les protocoles réseaux pour envoyer des paquets IP d'un ordinateur à un autre et pour envoyer un accusé de réception du destinataire à l'expéditeur. Tant que l'accusé de réception n'arrive pas, le même paquet est renvoyé périodiquement. Si trop d'accusés de réception n'arrivent pas, TCP ralentit la cadence d'envoi des paquets pour s'adapter à une congestion éventuelle du réseau global, puis ré-accélère la cadence quand les accusés de réception arrivent.

ALLER PLUS LOIN Le port

Comme plusieurs programmes peuvent utiliser TCP en même temps et sur la même machine, TCP attribue à chacun d'eux un numéro : *un port*. Un numéro de port permet à TCP de distinguer les paquets correspondant aux différents programmes qui communiquent en même temps.

Une autre fonction de TCP est de découper les informations à transmettre en paquets de 1 500 octets. Par exemple, pour envoyer une page web de 10 000 octets, TCP découpe ces 10 000 octets en paquets plus petits, chacun de taille standard, et les envoie l'un après l'autre. À l'autre bout du réseau, quand tous ces paquets standards sont arrivés, TCP les remet dans l'ordre et ré-assemble leurs contenus pour reconstruire la page web.

Exercice 16.12

Taper la commande `ping www.google.fr` dans une fenêtre *terminal* et observer ce qui se passe. En déduire la valeur d'un temps d'attente adéquat après lequel TCP devrait considérer que l'accusé de réception d'un paquet envoyé à `www.google.fr` est perdu. Quelle serait la conséquence d'utiliser un temps d'attente plus court ? Quelle serait la conséquence d'utiliser un temps d'attente plus long ?



Exercice 16.13

On suppose que la durée à attendre entre l'envoi d'un paquet et la réception d'un accusé de réception pour ce paquet est 1 seconde en moyenne, et que les paquets peuvent chacun contenir jusqu'à 1 500 octets de données. On considère les deux configurations suivantes pour TCP.

- Configuration A. TCP est configuré pour n'envoyer un nouveau paquet qu'après avoir reçu l'accusé de réception du paquet précédent.
- Configuration B. TCP est autorisé à envoyer des grappes de 20 paquets à la suite et à accuser réception avec un seul accusé de réception pour toute la grappe, au lieu d'un accusé de réception pour chaque paquet. En conséquence, un paquet perdu entraîne l'absence d'accusé de réception pour la grappe à laquelle il appartient, et donc demande de renvoyer la grappe entière au lieu du seul paquet perdu.

1. Combien de temps faut-il au minimum pour envoyer 1 Mo à destination avec la configuration A ? Avec la configuration B ? Pour simplifier, on suppose que le temps passé à envoyer 20 paquets à la suite est négligeable par rapport à 1 seconde.
2. On suppose maintenant qu'en moyenne, 1% des paquets envoyés vers une destination se perdent en chemin. En moyenne, combien de paquets faudra-t-il faire transiter sur le réseau pour transférer le fichier de 1 Mo à destination avec la configuration A ? Avec la configuration B ? Pour simplifier, on considère qu'à la deuxième fois qu'on envoie un paquet, il arrive à destination à coup sûr, et qu'un accusé de réception envoyé arrive également à coup sûr.
3. Quels sont les avantages et inconvénients de la configuration B par rapport à la configuration A ?



Exercice 16.14

Considérons un protocole de transport qui identifie chaque paquet envoyé par un numéro de séquence exprimé sur 4 octets, et admettons que chaque paquet envoyé peut contenir jusqu'à 1500 octets de données à transmettre. Quelle est la taille minimale de fichier à transmettre à partir de laquelle on devrait réutiliser un numéro de séquence déjà utilisé au début de l'envoi de ce même fichier ? Même question si on déduit des 1500 octets les informations de contrôle nécessaires au fonctionnement des couches au-dessus de la couche Lien, à savoir 20 octets d'en-tête pour le protocole de transport, et 20 octets d'en-tête IP ?

Exercice 16.15

Chercher sur le Web ce qu'est le protocole UDP. Quelles sont les différences entre UDP et TCP ? Quels sont les principaux avantages et inconvénients de chacun de ces protocoles ? Citer des exemples de programmes qui utilisent UDP, d'autres qui utilisent TCP, et expliquer ces choix.

Programmes utilisant le réseau : la couche application

Les protocoles des couches physique, lien, réseau et transport fournissent le socle d'Internet : ils permettent de transmettre de manière fiable des fichiers de toutes tailles, d'une machine à n'importe quelle autre machine connectée à Internet. En plus de ce socle, on distingue néanmoins un dernier type de protocoles, qui utilisent les services de la couche transport pour le compte de certains programmes que l'on utilise tous les jours, comme les navigateurs web ou les logiciels de courrier électronique. Il s'agit des *protocoles d'application*.

Les logiciels de courrier électronique utilisent par exemple le protocole d'application SMTP (S*imple Mail Transfer Protocol*), les navigateurs web utilisent le protocole d'application HTTP (*HyperText Transfer Protocol*) etc. Un autre protocole d'application important est DNS (*Domain Name System*) qui, aux adresses IP, associe des *noms de domaines* comme www.moi.fr.

SUJET D'EXPOSÉ DNS

Présenter les principes de base de DNS.

Quand un navigateur cherche à accéder à une page web située sur un autre ordinateur, il utilise DNS pour trouver l'adresse IP de l'ordinateur hôte sur lequel cette page web se trouve, puis le protocole HTTP pour demander cette page à l'ordinateur hôte. Si cette page est par exemple la page d'accueil d'un annuaire électronique, elle contiendra des champs à remplir, et c'est à nouveau le protocole HTTP qui acheminera les infor-

mations renseignées vers l'ordinateur hôte, qui, en fonction de ces informations, renverra en général une autre page avec la réponse à la requête.

Au bout du compte, pour transférer une page web d'un ordinateur à un autre, HTTP confie cette page web au protocole TCP, qui la découpe en paquets et confie chaque paquet au protocole IP, qui choisit un lien à utiliser en direction de la destination, puis confie chaque paquet au protocole de lien en vigueur sur ce dernier, par exemple WiFi, qui le confie enfin au protocole physique, qui gère l'acheminement des bits codant ces paquets à travers ce lien. Chaque protocole, à son niveau, contribue à la communication. Chaque protocole est simple ; c'est de leur interaction qui naît la complexité.

Quelles lois s'appliquent sur Internet ?

Jusqu'au milieu du XXe siècle, quand un livre ou un journal était publié, il l'était dans un pays particulier et sa publication était régie par les lois de ce pays. Quand un objet était vendu, il l'était dans un pays particulier et cette vente était régie par les lois de ce pays. Ainsi, la publication de certains textes ou la vente de certains objets était autorisée dans certains pays, mais interdite dans d'autres.

Parce que c'est un réseau mondial, Internet permet de publier, dans un pays, des textes qui peuvent être lus dans le monde entier et, de même, de vendre des objets qui peuvent être achetés dans le monde entier. Dès lors, quelle loi appliquer ? À cette question, qui est nouvelle, plusieurs réponses ont été imaginées, sans que personne ne sache encore laquelle s'imposera sur le long terme :

- l'application des lois du pays dans lequel le texte est publié ou l'objet mis en vente,
- l'application des lois du pays dans lequel le texte peut être lu ou l'objet acheté – ce qui obligerait, par exemple, un hébergeur de site web à bloquer l'accès à certains sites depuis certains pays,
- ou l'émergence d'un minimum de règles universelles.

Sujet d'exposé

L'affaire LICRA contre Yahoo!

Qu'est-ce que l'article R.645-1 du Code Pénal français ? Qu'est-ce que le premier amendement de la Constitution des États-Unis ? En quoi sont-ils contradictoires ? Chercher des documents sur l'*affaire de la LICRA contre Yahoo!*, relative à la vente aux enchères d'objets nazis sur Internet (de l'ordonnance du Tribunal de Grande Instance de Paris du 20 novembre 2000, jusqu'à la décision de la Cour Suprême des États-Unis du 30 mai 2006). Montrer en quoi cette affaire illustre le problème de l'application de législations différentes selon les pays pour la vente sur Internet.

Qui gouverne Internet ?

Quelques règles d'organisation d'Internet doivent être universellement acceptées. Par exemple, pour que les ordinateurs du monde entier puissent communiquer, il est nécessaire qu'ils utilisent les mêmes protocoles et pour qu'il n'y ait pas de confusion entre les adresses IP, il faut que deux ordinateurs distincts n'aient jamais la même adresse. Internet n'est donc pas entièrement décentralisé : un petit nombre de décisions doivent être prises en commun.

Ici, plusieurs modes d'organisation sont en concurrence, à nouveau sans que personne ne sache lequel s'imposera sur le long terme :

- l'émergence d'organisations internationales régies par des traités entre États,
- l'émergence d'organisations internationales informelles, dont la légitimité vient uniquement de la confiance qui leur est accordée,
- l'émergence d'organisations propres aux pays où Internet est le plus développé (cette dernière solution ayant l'inconvénient d'augmenter les différences de développement d'Internet entre les pays).

SUJET D'EXPOSÉ **Que sont...**

- l'*Internet Engineering Task Force* (IETF),
- l'*Internet Corporation for Assigned Names and Numbers* (ICANN),
- l'*Internet Society* (ISOC),
- le *World Wide Web Consortium* (W3C),
- le *WhatWG*,
- et l'*Union Internationale des Télécoms* (UIT) ?

Quel est le rôle et quel est le statut de chacune de ces organisations ?

ALLER PLUS LOIN **Calculer dans les nuages (*cloud computing*)**

Au cours de l'histoire de l'informatique, des modes centralisatrices et décentralisatrices se sont succédées. Ainsi, jusqu'aux années 1970, les entreprises n'avaient qu'un seul ordinateur, auquel étaient connectés de nombreux terminaux, par exemple formés d'un clavier et d'un écran, qui permettaient à différentes personnes d'effectuer des calculs sur cet ordinateur. À partir des années 1980, ces terminaux ont été remplacés par des micro-ordinateurs, connectés par un réseau à un serveur. Les calculs n'étaient alors plus effectués par un ordinateur central, mais par chacun de ces micro-ordinateurs.

Depuis le milieu des années 2000, on voit apparaître un retour de la centralisation. Des entreprises, qui vendaient naguère des programmes à des clients qui les utilisaient pour effectuer des calculs sur leurs ordinateurs, proposent désormais à ces mêmes clients d'effectuer elles-mêmes ces calculs à leur place. Les clients ont juste besoin de communiquer leurs données à ces entreprises, qui font les calculs sur leurs propres ordinateurs et envoient le résultat de ces calculs à leurs clients. C'est ce qu'on appelle le *calcul dans les nuages (cloud computing)*.

Par exemple, au lieu d'installer un logiciel de courrier électronique sur son ordinateur et de l'utiliser pour envoyer des courriers, on peut, à chaque fois que l'on souhaite envoyer un courrier, se connecter à un ordinateur distant, en général au moyen d'une page web, et communiquer le texte de son courrier à cet ordinateur, qui se chargera de l'envoyer ; c'est l'idée du *Webmail*. De même, au lieu d'acheter un logiciel de comptabilité et de l'utiliser, une entreprise peut se connecter, à chaque fois qu'elle souhaite effectuer une opération comptable, à une machine distante qui effectue cette opération pour l'entreprise. L'ordinateur local ne sert plus qu'à communiquer des informations à cette machine distante, comme jadis les terminaux.

Utiliser des programmes sur une machine distante simplifie beaucoup de choses. Il n'est plus nécessaire d'installer des logiciels sur son ordinateur, de les mettre à jour de temps en temps, etc. De plus, il devient possible d'envoyer un courrier depuis n'importe quel ordinateur : d'un web café de Bogota ou de Caracas, comme de son bureau. Pour une entreprise, cela permet de diminuer la taille du service informatique, puisqu'il lui suffit désormais d'avoir quelques ordinateurs reliés au réseau.

Cependant, cette évolution présente aussi un risque de dépossession des utilisateurs de leur pouvoir : au lieu d'avoir ses programmes, ses courriers, ses photos, sa comptabilité, etc. sur son ordinateur, on préfère les confier à des entreprises et des ordinateurs distants. En outre, on a parfois une garantie assez faible de leur conservation sur le long terme, de son pouvoir de les effacer ou de son contrôle sur les usages que ces entreprises peuvent faire de ces données.

Ai-je bien compris ?

- Qu'est-ce qu'un protocole ? Qu'est-ce qu'une couche ?
- Quelles sont les cinq couches de protocoles dont sont composés les réseaux ?
- Comment fabriquer un protocole fiable en utilisant un protocole peu fiable ?

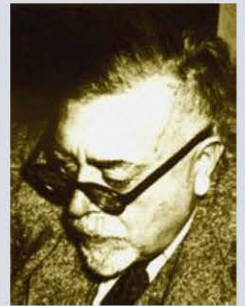
Les robots

17

CHAPITRE AVANCÉ

Un robot ? C'est un ordinateur à deux roues.

Dans ce chapitre, nous introduisons de nouveaux objets : les robots, qui sont essentiellement des ordinateurs munis de capteurs et d'actionneurs. Nous voyons comment les grandeurs captées sont numérisées et comment le principe de la boucle fermée permet de contrôler une action. Enfin, nous voyons comment programmer un robot à l'aide d'une boucle infinie dans laquelle les capteurs sont interrogés et les actionneurs activés.



Norbert Wiener (1894-1964) est le fondateur, à la fin des années 1940, de la science du pilotage ou *cybernétique*. Entouré d'un groupe interdisciplinaire de mathématiciens, logiciens, anthropologues, psychologues, économistes…, il a cherché à comprendre les processus de commande et de communication chez les êtres vivants, dans les machines et dans les sociétés. Le concept central de la cybernétique est celui de causalité circulaire ou contrôle en boucle fermée (*feedback*).

Les composants d'un robot



Comme un ordinateur ou un téléphone, un robot est formé d'un processeur, d'une mémoire et de périphériques. Ces derniers se divisent en périphériques de sortie, ou *actionneurs*, qui permettent au robot de se mouvoir et d'agir sur son environnement, et ses périphériques d'entrée, ou *capteurs*, qui lui permettent d'analyser cet environnement.

Dans ce chapitre, nous utilisons le robot mOway, mais les connaissances que nous présentons ne sont pas propres à ce robot et peuvent facilement se transposer à d'autres.

Les actionneurs du robot mOway sont deux moteurs qui font tourner ses roues et diverses diodes électroluminescentes que l'on peut allumer ou éteindre. Quand on fait tourner les deux roues du robot à la même vitesse, il avance en ligne droite. Quand on ralentit ou accélère une roue par rapport à l'autre, il tourne.

Les capteurs du robot mOway sont les suivants :

1


2


3

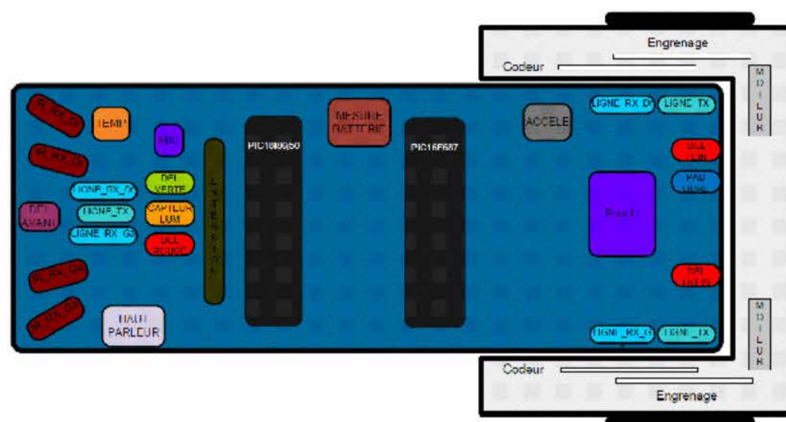

4


- Quatre détecteurs d'obstacles ❶ qui émettent régulièrement de brèves impulsions de lumière infrarouge. Quand le robot est proche d'un obstacle, cette lumière est réfléchie par l'obstacle et détectée par le robot.
- Un capteur de luminosité ❷, qui identifie la direction et l'intensité d'une source lumineuse.
- Deux capteurs de couleur de sol ❸, qui analysent la réflexion d'une lumière infrarouge sur le sol et permettent, par exemple, d'y repérer une ligne.
- Un capteur dont la résistance électrique varie avec la température.
- Un microphone qui détecte la présence ou l'absence d'un son dont la fréquence est comprise entre 100 Hz et 20 kHz, et aussi l'intensité de ce son.
- Un accéléromètre qui mesure l'accélération linéaire du robot et la gravité. C'est le même composant que sur les manettes de jeux et certains téléphones : une accélération déforme deux plaques souples d'un condensateur, ce qui fait varier sa capacité. Cet accéléromètre indique la direction verticale, ce qui permet, par exemple, de savoir si le robot s'est retourné.

Il est possible d'ajouter d'autres périphériques ❹, grâce à des bus similaires à ceux que nous avons décrits au chapitre 15.

Le robot contient une batterie rechargeable qui le rend autonome, mais il peut dialoguer avec un ordinateur par radio ou par un réseau WiFi (voir le chapitre 16).

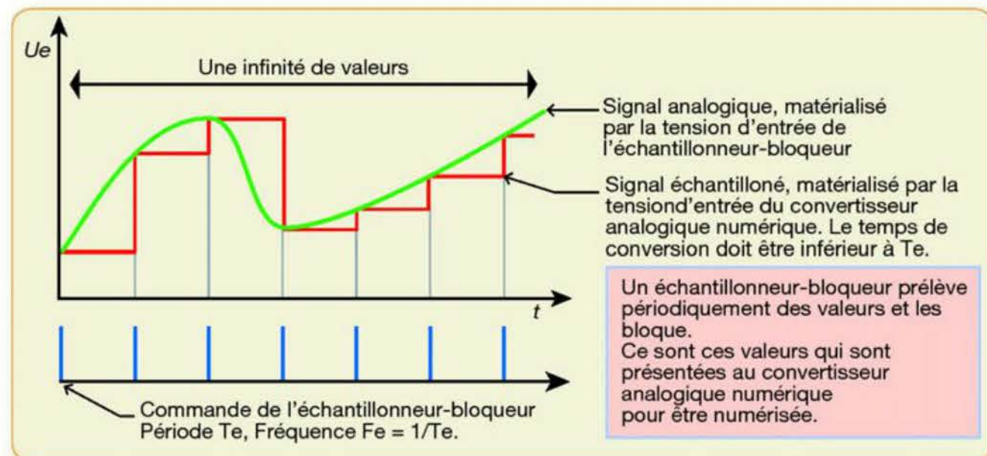
Tous les capteurs et actionneurs sont raccordés à deux *micro-contrôleurs*. Un micro-contrôleur est un circuit qui contient plusieurs composants, en particulier un processeur et de la mémoire (voir le chapitre 15). Le micro-contrôleur principal, appelé PIC18F86J50 sur la figure ci-après, exécute un programme chargé dans sa mémoire. Nous reviendrons plus tard sur le rôle du micro-contrôleur secondaire.



La numérisation des grandeurs captées

Les capteurs mesurent des grandeurs physiques : intensité lumineuse, intensité sonore, etc. et expriment en général ces grandeurs sous la forme d'une tension électrique. Pour que ces grandeurs puissent être utilisées par un processeur, cette tension doit, à son tour, être exprimée par un nombre représenté en binaire, comme le sont le niveau de gris d'un pixel quand on numérise une image ou la pression quand on numérise un son (voir le chapitre 9). Pour cela, deux composants sont utilisés : un *échantillonneur-bloqueur* et un *convertisseur analogique-numérique*. Le premier bloque la tension à une valeur stable, pendant que le second mesure cette valeur et produit un nombre, représenté en binaire, qui est la valeur de cette tension.

Par exemple, quand la valeur analogique est comprise entre 0 et 5 V et la valeur numérique comprise entre 0 et 1 023 (c'est-à-dire représentée sur 10 bits), la tension est mesurée par pas de 5 / 1 023 = 4,88 mV. Cette



valeur est appelée la *résolution* du convertisseur. Pour trouver la valeur numérique d'une tension, un convertisseur analogique-numérique n'utilise que des comparaisons entre cette tension et des multiples entiers de la résolution en procédant par dichotomie (voir le chapitre 20).

Exercice 17.1

Si un convertisseur analogique-numérique transforme une tension comprise entre 0 et 5 V en un nombre compris entre 0 et 1 023, à quelles valeurs analogiques correspondent les valeurs numériques 512, 256 et 768 ?

Exercice 17.2

Si un convertisseur analogique-numérique transforme une tension comprise entre 0 et 5 V en un nombre compris entre 0 et 1 023, en procédant par dichotomie, combien de comparaisons sont nécessaires pour déterminer la valeur numérique correspondante ?

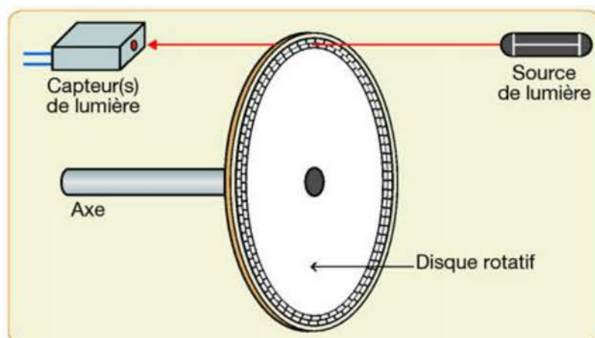
Exercice 17.3

Si on numérise une tension comprise entre 0 et 5 V, définie avec une précision de 20 mV, par un nombre compris entre 0 et 1 023, tous les bits de la valeur numérique ont-ils une signification ? Sur combien de bits suffirait-il de numériser cette valeur ?

Le contrôle de la vitesse : la méthode du contrôle en boucle fermée

Pour faire tourner un moteur à une vitesse déterminée, il ne suffit pas de fixer la tension d'alimentation du moteur, car la vitesse dépend aussi de la masse du robot, de la nature du terrain sur lequel le robot se déplace, des conditions climatiques si le robot est à l'extérieur, etc. La méthode appelée *contrôle en boucle fermée* utilise un capteur pour mesurer la vitesse du moteur, compare cette dernière à la vitesse souhaitée et réajuste la commande du moteur en fonction de l'écart constaté : si cette vitesse est inférieure à la vitesse souhaitée, on augmente la tension d'alimentation du moteur, si elle est supérieure, on la diminue. Mesurer en permanence la vitesse des moteurs et adapter leur tension d'alimentation en fonction de l'écart, par rapport à la consigne, est le rôle du micro-contrôleur secondaire, appelé PIC16F687 sur la figure.

Pour mesurer la vitesse de la roue du robot, on utilise un capteur de vitesse formé d'un disque qui alterne des zones opaques et transparentes, fixé sur l'axe du moteur et éclairé par une source de lumière. On calcule la vitesse du moteur en comptant le nombre de fois que la lumière est occultée par unité de temps. La fréquence de ce clignotement est proportionnelle à la vitesse. On utilise donc un circuit qui convertit cette fréquence en valeur de vitesse, de façon à pouvoir la contrôler.





Exercice 17.4

Pour le contrôleur de vitesse que l'on vient de décrire, quelle est la vitesse de la roue en fonction de la fréquence de clignotement, du nombre d'encoches sur la roue et du rayon de la roue ? Pour une roue de 2 cm de rayon et contenant 64 encoches, quelle vitesse correspond à la fréquence 128 Hz ? Quelle fréquence correspond à la vitesse 0,4 m/s ? Expliquer comment on peut exprimer la vitesse sous forme numérique, en utilisant un compteur numérique qui augmente d'une unité à chaque impulsion lumineuse.

Programmer un robot : les actionneurs

> **ALLER PLUS LOIN Un programme sans fin**
>
> Contrairement à beaucoup de programmes qui calculent un résultat et se terminent, un programme commandant un robot, un téléphone, un réseau et, plus généralement, un objet qui interagit avec son environnement, doit ne jamais se terminer, car le robot ou le téléphone ne cessent jamais d'interagir avec leur environnement. Quand un tel programme se termine, on dit que le robot ou le téléphone est *en panne*, car il ne répond plus aux sollicitations de son environnement et, bien souvent, on relance ce programme.

On programme le robot mOway en chargeant dans sa mémoire un programme, depuis un ordinateur ordinaire. Comme on l'a vu, ce programme est ensuite exécuté par le microcontrôleur principal. Ce programme doit être écrit, non en Java, mais en C. Cependant, le fragment de C, utilisé dans ce chapitre, est très similaire à Java.

On commence donc par écrire un programme et le compiler. On utilise pour cela l'environnement de développement *MPLAB*. On le transmet ensuite au robot à l'aide d'un câble USB et du programme *mOwayGUI* (*mOway Graphic User Interface*).

Pour écrire un programme, on utilise des fonctions qui interrogent les capteurs et commandent les actionneurs. Ces fonctions ne font pas partie du langage C lui-même, mais d'une extension de C fournie par le fabricant du robot. On indique que l'on utilise cette extension en ajoutant au début de son programme les commandes :

```
#include "lib_mot_moway.h"
#include "lib_sen_moway.h"
```

Pour faire avancer le robot, on utilise la fonction `MOT_STR`. Par exemple, l'instruction `MOT_STR(50,FWD,TIME,100);` fait avancer le robot à la vitesse 50, en marche avant, pendant 10 secondes (100 dixièmes de seconde).

Le premier argument de cette fonction est la vitesse à laquelle on fait avancer le robot. Il est compris entre 0 et 100 ; 0 correspond à 0 cm/s, 25 à 11 cm/s, 50 à 13 cm/s, 75 à 15 cm/s et 100 à 17 cm/s. Le deuxième, `FWD` ou `BACK`, définit le sens de la marche : avant ou arrière. Le troisième argument indique si l'on souhaite spécifier une durée ou une distance. Dans ce chapitre, nous spécifierons toujours une durée et cet argument sera `TIME`. Le quatrième argument, compris entre 0 et 255, est la durée du mouvement exprimée en dixièmes de secondes. On peut aussi faire avancer le robot pendant un temps infini en donnant, conventionnellement, la valeur 0 comme durée.

De même, pour faire tourner le robot, on utilise la fonction `MOT_ROT`. Par exemple, l'instruction `MOT_ROT(25,FWD,CENTER,LEFT,ANGLE,50);` fait tourner le robot à gauche, à la vitesse angulaire 25, d'un angle de 180 degrés (50 centièmes de tour).

Le premier argument de cette fonction est la vitesse angulaire à laquelle on fait tourner le robot. Il est compris entre 0 et 100 ; 0 correspond à

0 tour/s, 25 à 0,52 tour/s, 50 à 0,62 tour/s, 75 à 0,73 tour/s et 100 à 0,80 tour/s. Les deux arguments suivants permettent de choisir si l'on fait tourner le robot autour de son centre ou autour de l'une de ses roues. Dans ce chapitre, nous le ferons toujours tourner autour de son centre et ces arguments seront toujours `FWD` et `CENTER`. Le quatrième argument, `LEFT` ou `RIGHT`, définit le sens de rotation du robot : vers la gauche ou vers la droite. Le cinquième argument peut prendre deux valeurs, `ANGLE` ou `TIME` ; il indique si l'on souhaite spécifier l'angle de rotation ou la durée de la rotation. Le sixième argument est, en fonction de la valeur du cinquième, l'angle de la rotation exprimé en centièmes de tour ou sa durée exprimée en dixièmes de secondes. Il est compris entre 0 et 100 dans le premier cas et entre 0 et 255 dans le second. À nouveau, on peut aussi faire tourner le robot pendant un temps infini, en donnant, conventionnellement, la valeur 0 comme durée.

Quand on exécute une telle instruction `MOT_STR` ou `MOT_ROT`, on initie un mouvement, puis on passe à l'instruction suivante. Le robot continue alors son mouvement jusqu'à la fin, à moins que ce mouvement ne soit interrompu par l'initiation d'un autre mouvement.

En effet, si la poursuite de l'exécution du programme initie un second mouvement, alors le premier mouvement est interrompu. Par exemple, quand on exécute l'instruction :

```
MOT_STR(50,FWD,TIME,100);
MOT_ROT(25,FWD,CENTER,LEFT,ANGLE,50);
```

on commence par exécuter l'instruction `MOT_STR(50,FWD,TIME,100);` qui initie un mouvement rectiligne de 10 s, puis on exécute tout de suite la seconde instruction `MOT_ROT(25,FWD,CENTER,LEFT,ANGLE,50);` qui interrompt le mouvement en cours et initie un mouvement de rotation d'un angle de 180 degrés. Le mouvement rectiligne est interrompu quelques fractions de secondes seulement après avoir été initié. Il n'est donc pas effectué.

Si on souhaite effectuer le mouvement rectiligne en entier, avant de passer à la rotation, on doit utiliser la variable `MOT_END` qui prend la valeur 0 (équivalent de `false` en C) quand le robot est en mouvement et la valeur 1 (équivalent de `true` en C) quand le robot est immobile. Ainsi, quand on exécute l'instruction :

```
MOT_STR(50,FWD,TIME,100);
while(!MOT_END){}
MOT_ROT(25,FWD,CENTER,LEFT,ANGLE,50);
```

on initie un mouvement rectiligne de 10 s, on attend que le robot redevienne immobile, c'est-à-dire que le mouvement rectiligne soit achevé, puis on initie le mouvement de rotation.

Pour allumer et éteindre la diode électroluminescente, située à l'avant du robot, on utilise les instructions `LED_FRONT_ON();` et `LED_FRONT_OFF();`. Pour faire clignoter la diode électroluminescente verte, située sur le robot, on utilise l'instruction `LED_TOP_GREEN_ON_OFF();` Ici, le fait d'allumer une diode n'interrompt pas le mouvement du robot, les deux actions sont effectuées en même temps.

Pour attendre quelques secondes entre deux instructions, on utilise la fonction `Delay10KTCYx`, par exemple l'instruction `Delay10KTCYx(200);` interrompt l'exécution du programme pendant 2 s. L'argument de cette fonction est le temps du délai exprimé en centièmes de seconde.

Enfin, il faut exécuter au début de chaque programme les instructions `SEN_CONFIG();` et `MOT_CONFIG();` pour configurer les capteurs et les moteurs.

Par exemple, le programme :

```
main () {
  SEN_CONFIG();
  MOT_CONFIG();
  Delay10KTCYx(200);
  LED_TOP_GREEN_ON_OFF();
  MOT_STR(50,FWD,TIME,100);
  while(!MOT_END) {}
  MOT_ROT(25,FWD,CENTER,LEFT,ANGLE,50);
  while(1) {}}
```

attend 2 s avant de commencer, fait clignoter la diode verte, fait avancer le robot pendant 10 s, le fait tourner de 180 degrés et boucle à l'infini, afin que le programme ne se termine pas. Dans ce cas, la non-terminaison est un peu artificielle et sert surtout à éviter, comme on l'a expliqué, que le programme soit relancé.

Exercice 17.5

Écrire un programme qui fait avancer le robot en marche avant à la vitesse 100 pendant 5 s, puis le fait reculer à la vitesse 15 pendant 2 s.

Programmer un robot : les capteurs

D'autres fonctions interrogent les capteurs. Par exemple, l'expression `SEN_OBS_DIG(OBS_CENTER_L)` prend la valeur 1 (`true`) quand le capteur avant gauche détecte un obstacle, et la valeur 0 sinon. De même, les expressions

SEN_OBS_DIG(OBS_CENTER_R), SEN_OBS_DIG(OBS_SIDE_L) et SEN_OBS_DIG(OBS_SIDE_R) renvoient des valeurs similaires pour les capteurs avant droit, côté gauche et côté droit respectivement.

De même, l'expression SEN_LINE_DIG(LINE_L) prend la valeur 0 si le capteur de couleur de sol situé sur la gauche du robot capte une couleur claire et 1 s'il capte une couleur sombre. Le fonctionnement est le même pour le capteur de couleur de sol, situé sur la droite du robot, avec l'expression SEN_LINE_DIG(LINE_R).

À la différence des actionneurs que l'on commande quand on le souhaite, il faut interroger les capteurs de manière régulière, afin d'être prévenu de tous les événements qu'ils détectent. Une méthode pour ce faire est d'organiser le programme sous la forme d'une grande boucle qui, à chaque tour, interroge les capteurs et commande les actionneurs.

Le programme suivant par exemple fait clignoter la diode verte, puis initie un mouvement du robot en marche avant pour un temps infini. Si jamais le robot détecte un obstacle, on allume la diode située à l'avant du robot, on initie un mouvement de rotation de 180 degrés, ce qui interrompt le mouvement rectiligne, puis quand le mouvement de rotation est achevé, on relance le robot en marche avant pour un temps infini. Sinon, on éteint la diode située à l'avant du robot, si jamais elle est allumée.

```
void main() {
  SEN_CONFIG();
  MOT_CONFIG();
  LED_TOP_GREEN_ON_OFF();
  MOT_STR(50,FWD,TIME,0);
  while (1) {
    if (SEN_OBS_DIG(OBS_CENTER_L)){
      LED_FRONT_ON();
      MOT_ROT(25,FWD,CENTER,LEFT,ANGLE,50) ;
      while(!MOT_END){}
      MOT_STR(50,FWD,TIME,0);}
    else{
      LED_FRONT_OFF();}}}
```

SAVOIR-FAIRE Écrire un programme pour commander un robot

- Identifier les actionneurs et les capteurs à utiliser.
- Écrire les tests sur les valeurs des capteurs et les instructions initiant les actions dans une grande boucle infinie.
- Insérer les temporisations permettant aux actions de s'effectuer complètement si cela est nécessaire.
- Initialiser les capteurs et les actionneurs avant le début de la boucle.

Exercice 17.6 (avec corrigé)

Écrire un programme qui pilote un robot le long d'une ligne sombre dessinée sur un sol clair.

Les actionneurs utilisés sont, bien entendu, les moteurs que l'on commande avec instructions `MOT_STR` *et* `MOT_ROT`*. Les capteurs utilisés sont les capteurs de couleur de sol que l'on interroge avec les instructions* `SEN_LINE_DIG(LINE_L)` *et* `SEN_LINE_DIG(LINE_R)`*.*

On cherche à ce que le robot suive le côté gauche de la ligne, c'est-à-dire que son capteur gauche soit à l'extérieur de la ligne et son capteur droit à l'intérieur. 1

Tant que cela est le cas, le robot avance tout droit. En revanche, si les deux capteurs sont hors de la ligne, cela signifie que le robot est trop à gauche : il doit tourner à droite. Si les deux capteurs sont sur la ligne, le robot est trop à droite : il doit tourner à gauche. Si le capteur gauche est à l'intérieur de la ligne et le capteur droit à l'extérieur, cela signifie que le robot est dans le mauvais sens, il doit faire demi-tour. On utilise ici la même méthode que pour le contrôle de la vitesse du moteur : la méthode de contrôle en boucle fermée.

Il est important de placer le robot sur le bord de la ligne au moment où on le lance, sinon il ne fera que tourner sur lui-même.

```
void main () {
  Delay10KTCYx(200);
  SEN_CONFIG();
  MOT_CONFIG();
  while(1) {
    if (SEN_LINE_DIG(LINE_L)==0 && SEN_LINE_DIG(LINE_R)==1) {
      MOT_STR(80,FWD,TIME,0);}
    else {
      if (SEN_LINE_DIG(LINE_L)==0 && SEN_LINE_DIG(LINE_R)==0) {
        MOT_ROT(50,FWD,CENTER,RIGHT,TIME,0);}
      else {
        if (SEN_LINE_DIG(LINE_L)==1 && SEN_LINE_DIG(LINE_R)==1) {
          MOT_ROT(50,FWD,CENTER,LEFT,TIME,0);}
        else {
          MOT_ROT(50,FWD,CENTER,RIGHT,TIME,0);}}}}}
```



Exercice 17.7

Mettre le robot dans une enceinte avec un mur carré de 50 cm de côté, où il n'y a pas d'autres obstacles que les murs. Utiliser le détecteur d'obstacles pour le faire aller vers l'un des murs, puis tourner sans fin dans le sens des aiguilles d'une montre.



Exercice 17.8

Dans un espace sans limite avec deux robots face à face, écrire un programme qui fait danser ensemble les robots en les faisant tourner l'un autour de l'autre.



Exercice 17.9

On imagine un robot aspirateur dans une enceinte avec un mur carré de 50 cm de côté, où il n'y a pas d'autres obstacles que les murs. Écrire un programme pour que le robot passe l'aspirateur sur tout le sol. Essayer différentes méthodes : par des allers-retours, en spirale, etc. Essayer ces programmes.

ALLER PLUS LOIN Les interruptions

Organiser un programme en une grande boucle qui teste les capteurs en permanence est possible, mais souvent malcommode. On a donc introduit dans les langages de programmation des outils qui permettent d'exprimer la même chose de manière plus simple. Le programme décrit d'une part ce qu'il faut faire quand tout se passe normalement, par exemple avancer tout droit, et d'autre part des conditions qui définissent des *interruptions*, par exemple le fait qu'un détecteur signale un obstacle, et des instructions à exécuter en cas d'interruption, par exemple faire tourner le robot. Ces différentes instructions sont ensuite traduites automatiquement en un programme qui, de manière répétée, interroge les capteurs et, selon qu'une interruption est déclenchée ou non, exécute une instruction ou une autre.
Cette manière *réactive* de programmer, permet de mieux prendre en compte les aléas de l'environnement : d'une exécution d'un programme à une autre, un robot rencontre rarement deux fois la même situation et il doit s'adapter s'il rencontre une tache d'huile sur le sol, des obstacles nouveaux, etc.

ALLER PLUS LOIN Les mots « robot » et « robotique »

Le mot « robot », dérivé d'un mot qui signifie « esclave », a été créé par l'écrivain Tchèque Karel Capek en 1920 et le mot « robotique » par un autre écrivain, Isaac Asimov, en 1942. L'origine littéraire de ces deux mots n'est pas due au hasard. Le fait que les ordinateurs imitent certaines facultés humaines, comme effectuer des multiplications ou jouer aux échec, a suscité le rêve de machines intelligentes. Mais la conception de robots, c'est-à-dire d'ordinateurs mobiles et autonomes, rejoint, de plus, une figure littéraire ancienne, qui du Golem à Pinocchio et du monstre de Frankenstein à WALL-E, pose la question de la frontière entre l'animé et l'inanimé.

ALLER PLUS LOIN Les robots sont partout

Dans l'industrie, les robots sont utilisés dans les chaînes de montage de nombreuses usines. En médecine, ils sont utilisés pour effectuer des opérations chirurgicales micro-invasives, pour effectuer des analyses, pour remplacer des membres paralysés, pour assister des personnes dépendantes, etc. Ils sont aussi utilisés dans l'exploration spatiale et sous-marine ou pour intervenir dans les zones inaccessibles, par exemple de centrales nucléaires. Des robots sont aussi utilisés dans des tâches plus quotidiennes comme passer l'aspirateur, nettoyer une piscine ou garer une voiture.



Thyroïdectomie assistée par un robot - CHU de Nîmes



Le robot industriel *Robolab* recopiant la Bible

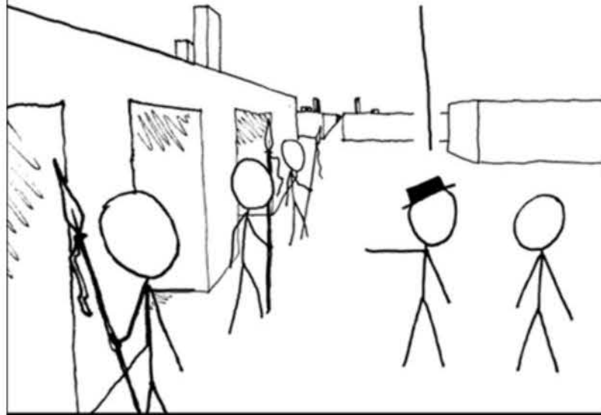
Ai-je bien compris ?

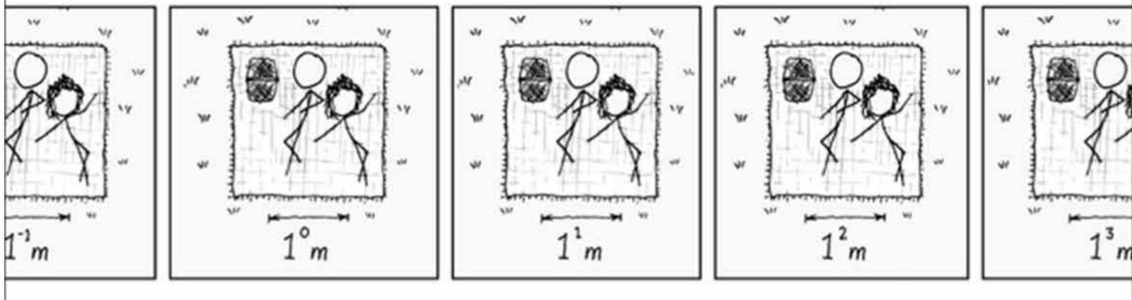
- Quels sont les composants d'un robot ?
- Qu'est-ce que le principe de la boucle fermée ?
- Comment organise-t-on un programme pour que les capteurs soient interrogés périodiquement ?

Quatrième partie

Algorithmes

Dans cette quatrième partie, nous apprenons quelques-uns des savoir-faire les plus utiles au XXIe siècle : ajouter des nombres exprimés en base deux (chapitre 18), dessiner (chapitre 19), retrouver une information par dichotomie (chapitre 20*), trier des informations (chapitre 21*) et parcourir un graphe (chapitre 22*).


ET LÀ, CE SONT LES GARDIENS DU LABYRINTHE.
L'UN D'EUX MENT TOUJOURS, UN AUTRE DIT TOUJOURS
LA VÉRITÉ, ET LE DERNIER FOUDROIE TOUS CEUX
QUI POSENT DES QUESTIONS TROP SUBTILES.


PUISSANCES DE UN
UNE AUTRE VISION DU MONDE.
1^{-1}m
1^{0}m
1^{1}m
1^{2}m
1^{3}m

XKCD

Ajouter deux nombres exprimés en base deux

18



Ada Lovelace (1815-1852) est l'auteur du premier algorithme destiné à être exécuté par une machine. Cet algorithme, qui permettait de calculer une suite de nombres de Bernouilli, devait être exécuté sur la machine analytique conçue par Charles Babbage. Malheureusement, Babbage n'a jamais réussi à terminer sa machine. Ada Lovelace est parfois considérée comme le premier programmeur de l'histoire. Le langage de programmation Ada est ainsi nommé en son honneur.

Pour faire une addition, l'ordinateur fait comme on lui a appris sur les bancs de l'école.

Dans ce chapitre, nous détaillons l'algorithme de l'addition en base deux, ce qui est surtout un prétexte pour comprendre comment démontrer qu'un algorithme est correct.

L'algorithme est le même que celui que nous utilisons couramment lorsque nous effectuons une addition ordinaire, c'est-à-dire en base dix. Nous démontrons ensuite que l'algorithme que nous avons programmé calcule bien la somme de deux nombres. Pour cela, nous utilisons la notion importante d'invariant de boucle qui est une propriété vraie à chaque tour de boucle. Nous montrons une telle propriété par récurrence sur le numéro du tour de boucle.

Nous revenons dans ce chapitre sur l'un des premiers algorithmes que nous ayons appris à l'école, celui qui permet d'ajouter deux nombres entiers, pour nous poser deux questions : comment adapter cet algorithme aux nombres exprimés en base deux ? Et pourquoi cet algorithme calcule-t-il bien la somme des deux nombres ?

L'addition

Commençons par rappeler cet algorithme sur un exemple. On veut ajouter les nombres 728 et 456.

```
 1010
  728
  456
 ----
 1184
```

On commence par ajouter les chiffres des unités, 8 et 6. La table de l'addition indique que la somme de ces deux chiffres est 14 ; on pose le chiffre des unités, 4, et on retient le chiffre des dizaines, 1. On ajoute ensuite les chiffres des dizaines et cette retenue, 2, 5 et 1. La table de l'addition indique que la somme de ces trois chiffres est 8 ; on pose le chiffre des unités, 8, et on retient le chiffre des dizaines, 0. On ajoute ensuite les chiffres des centaines et cette retenue, 7, 4 et 0. La table de l'addition indique que la somme de ces trois chiffres est 11 ; on pose le chiffre des unités, 1, et on retient le chiffre des dizaines, 1. Finalement, on pose cette retenue dans la colonne des milliers.

Une irrégularité de cette méthode est que, lors de la première itération, on ajoute deux chiffres, alors qu'en régime permanent, on en ajoute trois. On peut corriger cela en commençant par poser la retenue égale à 0. La première itération se formule alors de la manière suivante : on commence par ajouter les chiffres des unités et la retenue, 8, 6 et 0, etc. Ainsi, cet algorithme n'utilise qu'une seule table, qui indique la somme de chacun des triplets $(a ; b ; c)$ où a, b et c sont des chiffres compris entre 0 et 9, table qu'en général on connaît par cœur.

L'addition pour les nombres exprimés en base deux

Voyons maintenant comment on ajoute des nombres exprimés en base deux, par exemple 101 et 111, c'est-à-dire 5 et 7.

```
1 1 1 0
  1 0 1
  1 1 1
-------
1 1 0 0
```

On commence, comme en base dix, par ajouter les chiffres des unités et la retenue, 1, 1 et 0. La table de l'addition indique que la somme de ces trois chiffres est 10 ; on pose le chiffre des unités, 0, et on retient le chiffre des deuzaines, 1. On ajoute ensuite les chiffres des deuzaines et cette retenue, 0, 1 et 1. La table de l'addition indique que la somme de ces trois chiffres est 10 ; on pose le chiffre des unités, 0, et on retient le chiffre des deuzaines, 1. On ajoute ensuite les chiffres des quatraines et cette retenue, 1, 1 et 1. La table de l'addition indique que la somme de ces trois chiffres est 11 ; on pose le chiffre des unités, 1, et on retient le chiffre des deuzaines, 1. Finalement, on pose cette retenue dans la colonne des huitaines. Le résultat est donc 1100, c'est-à-dire 12.

Cette méthode utilise une table qui indique la somme de chacun des triplets (*a* ; *b* ; *c*) où *a*, *b* et *c* sont des chiffres compris entre 0 et 1. Cette table ne contient donc que huit lignes :

a	b	c	a + b + c
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	10
1	0	0	1
1	0	1	10
1	1	0	10
1	1	1	11

En fait, cette méthode se formule mieux en utilisant deux tables. La première indique le chiffre des unités de $a + b + c$:

a	b	c	Unités de a + b + c
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

La seconde indique le chiffre des deuzaines de $a + b + c$:

a	b	c	Deuzaines de a + b + c
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Il s'agit là des tables de deux fonctions booléennes à trois arguments. Comme toutes les fonctions booléennes, elles peuvent s'exprimer avec les fonctions *non*, *et* et *ou* (voir le chapitre 10). Une manière, parmi d'autres, de les exprimer est la suivante :

```
unités(a,b,c)
= (a et non(b) et non(c)) ou (non(a) et b et non(c))
  ou (non(a) et non(b) et c) ou (a et b et c)
deuzaines(a,b,c) = (a et b) ou (b et c) ou (a et c)
```

Pour se convaincre de la correction de ces expressions, il suffit de vérifier qu'elles donnent bien les chiffres des unités et des deuzaines de $a + b + c$ dans chacun des huit cas des tables précédentes. Par exemple, dans le cas $a = 0$, $b = 1$ et $c = 1$, l'expression *a et non*(*b*) *et non*(*c*) prend la valeur 0, les expressions *non*(*a*) *et b et non*(*c*), *non*(*a*) *et non*(*b*) *et c* et *a et b et c* prennent elles aussi la valeur 0 et donc l'expression (*a et non*(*b*) *et non*(*c*)) *ou* (*non*(*a*) *et b* et *non*(*c*)) *ou* (*non*(*a*) *et non*(*b*) *et c*) *ou* (*a et b et c*) prend la

valeur 0 également, ce qui est bien le chiffre des unités de $a + b + c$. De même, les expressions *a et b*, *b et c* et *a et c* prennent respectivement les valeurs 0, 1 et 0 et donc l'expression (*a et b*) *ou* (*b et c*) *ou* (*a et c*) prend la valeur 1, ce qui est bien le chiffre des deuzaines de $a + b + c$.

On peut, par exemple, programmer cette méthode pour ajouter deux nombres de dix chiffres binaires. Le résultat sera donc un nombre de onze chiffres. On choisit de représenter les nombres à ajouter x et y par deux tableaux de booléens de dix cases `n` et `p`, et le résultat par un tableau de booléens `r` de 11 cases. On choisit le booléen `true` pour le chiffre 1 et le booléen `false` pour le chiffre 0. La case 0 d'un tableau contient le chiffre des unités du nombre représenté, la case 1 le chiffre des deuzaines… et la case 9, le chiffre des cinq-cent-douzaines. Le résultat `r` qui a un chiffre de plus a aussi une case 10 pour les mille-vingt-quatraines.

La retenue `c` est d'abord initialisée à 0 (❶). Puis on calcule les chiffres du résultat l'un après l'autre par une boucle dont l'indice `i` varie de 0 à 9. À chaque étape, on définit le chiffre `a` comme le `i`-ème chiffre du nombre `n` (❷) et `b` comme le `i`-ème chiffre du nombre `p` (❸), puis on affecte la case `i` du tableau `r` (❹) avec le chiffre des unités de `a + b + c`. Enfin, on affecte la retenue `c` avec le chiffre des deuzaines de `a + b + c` (❺). Et une fois la boucle terminée, on affecte la case 10 du tableau `r` avec la dernière des retenues (❻).

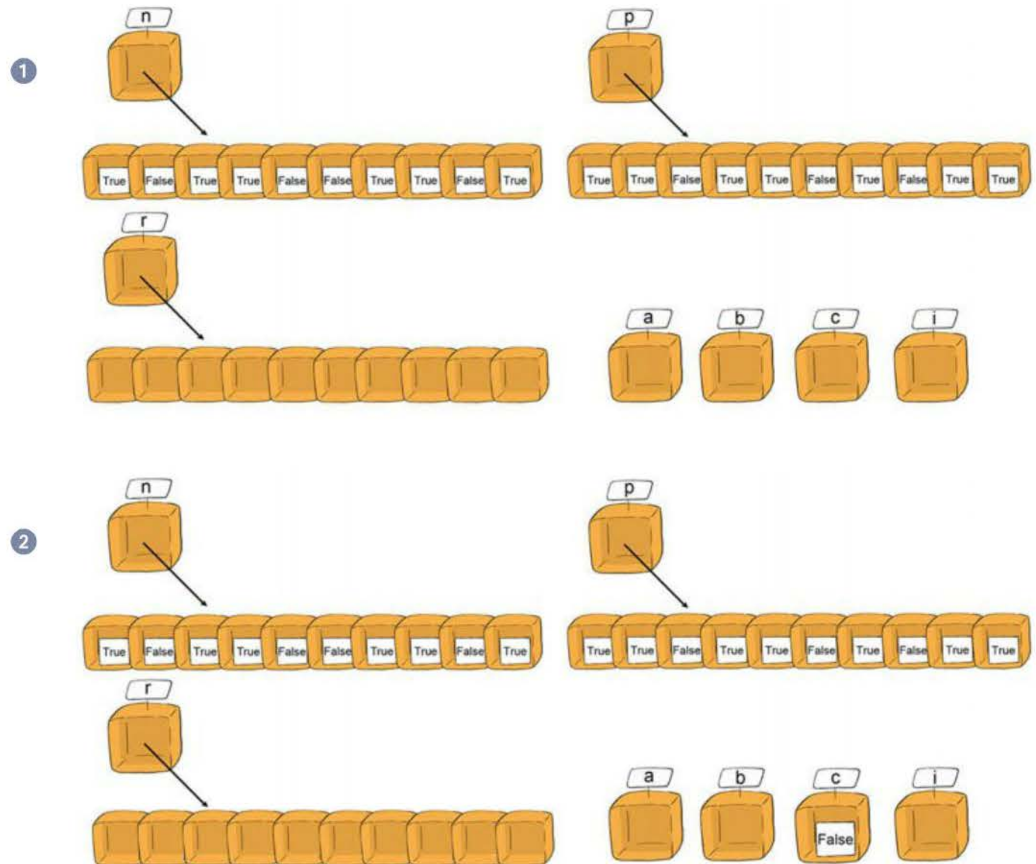
```
c = false;❶
for (i = 0; i <= 9; i = i + 1) {
  a = n[i];❷
  b = p[i];❸
  r[i] = (a && !b && !c) || (!a && b && !c) || (!a && !b && c)
         || (a && b && c);❹
  c = (a && b) || (b && c) || (a && c);}❺
r[10] = c;❻
```

Exercice 18.1

On utilise ce programme pour ajouter les nombres $x = \underline{1011001101}$ et $y = \underline{1101101011}$. Exécuter l'instruction `c = false;` l'initialisation de la variable `i` et le tour 0 de la boucle revient à exécuter la séquence d'affectations suivante :

```
c = false;
i = 0;
a = n[0];
b = p[0];
r[0] = (a && !b && !c) || (!a && b && !c) || (!a && !b && c)
       || (a && b && c);
c = (a && b) || (b && c) || (a && c);
i = i + 1;
```

Si on exécute cette séquence dans l'état ❶, la première affectation `c = false;` donne l'état ❷.



Dessiner les états successifs produits par l'exécution de chacune de ces affectations. Quel est l'état final produit par l'exécution du tour 0 de la boucle ?

Montrer que dans cet état :

$$(r[0] \times 2^0) + c \times 2^1 = (n[0] \times 2^0) + (p[0] \times 2^0)$$

exécuter le tour 1 de la boucle revient à exécuter la séquence d'affectations suivante :

```
a = n[1];
b = p[1];
r[1] = (a && !b && !c) || (!a && b && !c) || (!a && !b && c)
        || (a && b && c);
c = (a && b) || (b && c) || (a && c);
i = i + 1;
```

Quel est l'état produit par l'exécution de ce tour de boucle ?

Montrer que dans cet état :

$$(r[0] \times 2^0 + r[1] \times 2^1) + c \times 2^2 = (n[0] \times 2^0 + n[1] \times 2^1) + (p[0] \times 2^0 + p[1] \times 2^1)$$

La démonstration de correction du programme

Quand on conçoit un tel programme, une question se pose naturellement : comment sait-on qu'il calcule la somme des deux nombres entiers ?

Une première manière de s'assurer qu'un programme fait bien ce qu'on attend de lui est de le tester (voir le chapitre 1). Il faut essayer différentes valeurs pour les nombres x et y et vérifier que le programme affiche bien la valeur $x + y$ dans tous les cas. On estime, en général, que le coût du test d'un programme est du même ordre de grandeur que celui de son développement. Cependant, le test présente deux limites importantes : la première est que l'on ne peut pas tester le programme sur toutes les valeurs d'entrée possibles, qui sont souvent très nombreuses, voire en nombre infini. La seconde est que pour tester un programme, il faut savoir ce que l'on attend de lui. Or, ce n'est pas le cas quand on écrit, par exemple, un programme qui calcule la millième décimale du nombre π, ou la position de la Lune dans mille ans, car la raison pour laquelle on écrit un tel programme est précisément que l'on ignore la millième décimale du nombre π ou la position de la Lune dans mille ans. De ce fait, comment le tester ?

Une autre manière de s'assurer qu'un programme fait bien ce qu'on attend de lui est de le démontrer. Par exemple, on peut démontrer que le programme précédent calcule bien la somme des nombres x et y. Plus précisément, on veut démontrer que si, au moment où l'on exécute ce programme, les tableaux `n` et `p` contiennent la représentation binaire de deux entiers de dix chiffres, c'est-à-dire si :

$$x = n[0] \times 2^0 + n[1] \times 2^1 + \ldots + n[8] \times 2^8 + n[9] \times 2^9$$

et :

$$y = p[0] \times 2^0 + p[1] \times 2^1 + \ldots + p[8] \times 2^8 + p[9] \times 2^9$$

Invariant

Un *invariant* d'une boucle est une propriété qui est vérifiée à chaque exécution du corps de cette boucle. En général, pour la dernière exécution, cette propriété traduit le fait que la boucle réalise bien la tâche souhaitée. On montre qu'une propriété est un invariant d'une boucle par un raisonnement par récurrence :

- on montre que la propriété est vérifiée à la première exécution du corps de la boucle,
- on montre que si l'invariant est vérifié à une exécution donnée du corps de la boucle, il est encore vérifié à l'exécution suivante.

L'invariant est alors vérifié à la fin de boucle, qui fournit donc le résultat attendu.

alors à la fin de l'exécution de ce programme, le tableau `r` contient un nombre de onze chiffres qui est la représentation binaire de $x + y$, c'est-à-dire que :

> $x + y =$ `r[0]` $\times 2^0 +$ `r[1]` $\times 2^1 + \ldots +$ `r[9]` $\times 2^9 +$ `r[10]` $\times 2^{10}$.

Le programme qui ajoute deux nombres exprimés en binaire est formé d'une boucle, dans laquelle on calcule d'abord le chiffre des unités, puis les chiffres des deuzaines, des quatraines, etc. du résultat. Après avoir achevé les tours 0..., `i - 1` et au moment de commencer le tour `i`, on a donc calculé la somme des deux nombres formés des `i - 1` premiers chiffres, en partant de la droite, des nombres x et y. C'est l'invariant que l'on va montrer par récurrence. À la fin de la boucle, l'invariant indiquera que l'algorithme a effectué l'addition souhaitée.

Par exemple, si au cours de l'addition de $x =$ 1011001101 et $y =$ 1101101011 (❶) on s'arrête après avoir effectué les tours 0 et 1 de la boucle et avant de commencer le tour 2, on a déjà calculé la somme des nombres 01 et 11, c'est-à-dire 1 et 3 ❷. Le résultat de cette addition n'est pas exactement le nombre représenté par les deux chiffres déjà posés 00, car il faut tenir compte de la retenue. La propriété exacte est que si on pose la retenue dans la colonne `i`, ce qui donne dans cet exemple le nombre 100 c'est-à-dire 4, on obtient la somme des deux nombres formés des `i - 1` premiers chiffres des nombres x et y.

❶

```
1 1 1 1 0 0 1 1 1 1
  1 0 1 1 0 0 1 1 0 1
  1 1 0 1 1 0 1 0 1 1
---------------------
1 1 0 0 0 1 1 1 0 0 0
```

❷

```
                 1 1
1 0 1 1 0 0 1 1 0 1
1 1 0 1 1 0 1 0 1 1
-------------------
                 0 0
```

Autrement dit, au moment de commencer le tour `i` de la boucle, l'état vérifie la propriété :

> (`r[0]` $\times 2^0 + \ldots +$ `r[i-1]` $\times 2^{i-1}$) $+$ `c` $\times 2^i$
> $=$ (`n[0]` $\times 2^0 + \ldots +$ `n[i-1]` $\times 2^{i-1}$) $+$ (`p[0]` $\times 2^0 + \ldots +$ `p[i-1]` $\times 2^{i-1}$)

On démontre maintenant cette propriété.

À la première exécution, `i` = 0, la somme (`r[0]` $\times 2^0 + \ldots +$ `r[i-1]` $\times 2^{i-1}$) ne contient aucun terme ; elle vaut donc 0. Il en est de même pour les sommes (`n[0]` $\times 2^0 + \ldots +$ `n[i-1]` $\times 2^{i-1}$) et (`p[0]` $\times 2^0 + \ldots +$ `p[i-1]` $\times 2^{i-1}$). Comme par ailleurs, la retenue `c` vaut 0, les deux membres de l'égalité sont nuls.

On suppose maintenant que cette propriété est vérifiée dans l'état dans lequel s'exécute le tour `i` de la boucle et on veut montrer qu'elle est encore vérifiée dans l'état dans lequel s'exécute le tour suivant. Au début du tour `i`, on a :

$$(\texttt{r[0]} \times 2^0 + \ldots + \texttt{r[i-1]} \times 2^{i-1}) + \texttt{c} \times 2^i$$
$$= (\texttt{n[0]} \times 2^0 + \ldots + \texttt{n[i-1]} \times 2^{i-1}) + (\texttt{p[0]} \times 2^0 + \ldots + \texttt{p[i-1]} \times 2^{i-1})$$

et donc en ajoutant $\texttt{n[i]} \times 2^i + \texttt{p[i]} \times 2^i$ dans les deux membres de l'égalité on obtient que, au début du tour `i` de la boucle :

$$(\texttt{r[0]} \times 2^0 + \ldots + \texttt{r[i-1]} \times 2^{i-1}) + (\texttt{n[i]} + \texttt{p[i]} + \texttt{c}) \times 2^i$$
$$= (\texttt{n[0]} \times 2^0 + \ldots + \texttt{n[i]} \times 2^i) + (\texttt{p[0]} + \ldots + \texttt{p[i]} \times 2^i)$$

Au cours de ce tour de la boucle, on ajoute les trois chiffres `n[i]`, `p[i]` et `c`, et le résultat de cette addition a pour chiffre des unités `r[i]` et pour chiffre des deuzaines la nouvelle valeur de `c`, si bien que, dans l'état atteint à la fin de ce tour de la boucle :

$$(\texttt{r[0]} \times 2^0 + \ldots + \texttt{r[i-1]} \times 2^{i-1}) + (\texttt{r[i]} + 2 \times \texttt{c}) \times 2^i$$
$$= (\texttt{n[0]} \times 2^0 + \ldots + \texttt{n[i]} \times 2^i) + (\texttt{p[0]} \times 2^0 + \ldots + \texttt{p[i]} \times 2^i)$$

c'est-à-dire :

$$(\texttt{r[0]} \times 2^0 + \ldots + \texttt{r[i]} \times 2^i) + \texttt{c} \times 2^{i+1}$$
$$= (\texttt{n[0]} \times 2^0 + \ldots + \texttt{n[i]} \times 2^i) + (\texttt{p[0]} \times 2^0 + \ldots + \texttt{p[i]} \times 2^i)$$

Au début du tour suivant, la variable `i` a été augmentée de 1, si bien que :

$$(\texttt{r[0]} \times 2^0 + \ldots + \texttt{r[i-1]} \times 2^{i-1}) + \texttt{c} \times 2^i$$
$$= (\texttt{n[0]} \times 2^0 + \ldots + \texttt{n[i-1]} \times 2^{i-1}) + (\texttt{p[0]} \times 2^0 + \ldots + \texttt{p[i-1]} \times 2^{i-1})$$

La propriété est donc encore vérifiée au début du tour suivant. Elle est donc vérifiée à chacun des tours de boucles : c'est un invariant de la boucle.

À la fin du dernier tour, `i` est égal à 10 et donc :

$$(\texttt{r[0]} \times 2^0 + \ldots + \texttt{r[9]} \times 2^9) + \texttt{c} \times 2^{10}$$
$$= (\texttt{n[0]} \times 2^0 + \ldots + \texttt{n[9]} \times 2^9) + (\texttt{p[0]} \times 2^0 + \ldots + \texttt{p[9]} \times 2^9)$$
$$= x + y$$

On affecte alors la case 10 du tableau `r` avec la retenue si bien que, quand l'exécution est terminée :

$$\texttt{r[0]} \times 2^0 + \ldots + \texttt{r[10]} \times 2^{10} = x + y$$

C'est ce qu'il fallait démontrer : le tableau `r` contient la représentation binaire du nombre $x + y$.

ALLER PLUS LOIN L'autonomie de la notion d'algorithme

Dans ce chapitre, nous avons étudié un programme, écrit en Java, qui additionne deux nombres écrits en base deux. Il est possible d'écrire des programmes très similaires dans d'autres langages de programmation. La méthode pour ajouter deux nombres en base deux est indépendante d'un langage de programmation particulier : c'est une méthode abstraite qui peut s'exprimer dans divers langages. Une telle méthode systématique qui permet de résoudre un problème s'appelle un *algorithme*. Il est important de distinguer un algorithme, méthode indépendante de tout langage, d'un *programme*, qui est l'incarnation d'un algorithme dans un langage particulier.

Cette distinction entre les notions d'algorithme et de programme doit également son importance au fait que nous ayons utilisé des algorithmes pour faire des additions dans diverses bases depuis des millénaires, bien avant que nous ayons pensé à exprimer cet algorithme dans un langage de programmation. Nous avons même utilisé des algorithmes, transmis de génération en génération par observation et imitation, pour fabriquer des objets en céramique, tisser des étoffes, nouer des cordages, préparer les aliments, etc. avant l'invention de l'écriture.

ALLER PLUS LOIN Définitions algorithmiques et non algorithmiques

L'apprentissage des mathématiques commence par l'apprentissage d'algorithmes qui permettent d'effectuer des additions, des soustractions, etc.
Même au-delà de ces mathématiques élémentaires, beaucoup de définitions mathématiques sont algorithmiques. Par exemple, la définition du nombre $n - m$ comme le nombre obtenu en mettant n cailloux dans un sac, en en ôtant m et en comptant ceux qui restent est algorithmique. Mais la définition du nombre $n - m$ comme le nombre p tel que $p + m = n$ ne l'est pas : contrairement à la première, cette définition ne dit pas ce que l'on doit faire pour connaître le nombre $n - m$ quand on connaît les nombres n et m.

De même, la définition selon laquelle deux vecteurs non nuls du plan, donnés par leurs coordonnées $(x_1 ; y_1)$ et $(x_2 ; y_2)$ dans une base, sont colinéaires quand $x_1 y_2 = x_2 y_1$ est algorithmique, mais pas celle selon laquelle ces deux vecteurs sont colinéaires s'il existe un facteur de proportion k tel que $x_1 = k x_2$ et $y_1 = k y_2$. Si deux vecteurs sont donnés par leurs coordonnées dans une base, par exemple (4 ; 10) et (6 ; 15), la première définition donne une méthode pour déterminer s'ils sont colinéaires, puisqu'il suffit de calculer 4×15 et 10×6 et de vérifier que l'on obtient bien le même nombre dans les deux cas, mais pas la seconde, qui demande de trouver le facteur de proportion, sans indiquer de méthode pour le faire.

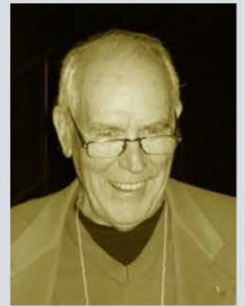
Ai-je bien compris ?

- En quelles bases l'algorithme de l'addition peut-il être utilisé ?
- Que veut-on dire lorsqu'on affirme que l'algorithme de l'addition est correct ?
- Qu'est-ce qu'un invariant ?

19 Dessiner

Ou comment devenir Botticelli sans se tacher les doigts.

Dans ce chapitre, nous voyons comment programmer un ordinateur pour dessiner ou modifier une image… sans utiliser un logiciel de retouche photo ! Nous voyons comment ouvrir une fenêtre graphique, créer une image, dessiner en trois dimensions, lire et produire des fichiers contenant des images, transformer des images.



Ivan Sutherland (1938-) est un des pionniers de l'informatique graphique. Il est l'auteur du logiciel *Sketchpad* (1963) qui est l'un des premiers logiciels de conception assistée par ordinateur. Ivan Sutherland a aussi été à l'origine de l'un des premiers systèmes de réalité virtuelle muni d'un visiocasque. Il est l'un des pionniers des architectures d'ordinateurs spécialisées pour le temps réel et le graphisme.

Dessiner dans une fenêtre

Deux instructions permettent d'ouvrir une *fenêtre graphique* (c'est-à-dire une fenêtre dans laquelle on peut dessiner) et d'y dessiner un pixel.

Exécuter l'instruction `Isn.initDrawing("Mon premier dessin",x,y,largeur,hauteur);` a pour effet d'ouvrir une fenêtre de `largeur` pixels de large, sur `hauteur` pixels de haut, qui porte le nom `Mon premier dessin` et dont le coin en haut à gauche est au pixel de coordonnées (`x` ; `y`) de l'écran.

> **ATTENTION Axe vertical**
>
> De même que les Anglais roulent à gauche, l'axe vertical, en géométrie algorithmique, est orienté vers le bas.

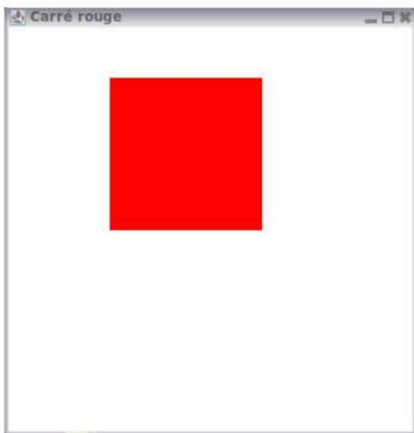
Exécuter l'instruction `Isn.drawPixel(x,y,rouge,vert,bleu);` a pour effet de dessiner un pixel dans la `x`-ème colonne et la `y`-ème ligne de cette fenêtre, dont la couleur est décrite par les nombres `rouge`, `vert` et `bleu` (voir le chapitre 9). La coordonnée `x` varie entre 0 et `largeur` - 1 et la coordonnée `y` entre 0 et `hauteur` - 1. Les nombres `rouge`, `vert` et `bleu` varient entre 0 et 255.

SAVOIR-FAIRE Créer une image

1. Établir une condition sur les coordonnées d'un pixel qui permette de décider s'il appartient ou non à la figure à tracer.
2. Écrire une instruction qui balaye la fenêtre graphique, au moyen de deux boucles imbriquées, l'une sur les abscisses et l'autre sur les ordonnées.
3. Dans le corps de la boucle la plus interne, affecter la couleur appropriée à chaque pixel, selon qu'il appartient ou non à la figure.

Exercice 19.1 (avec corrigé)

Dans une fenêtre de 400 pixels sur 400 pixels, dessiner un carré rouge formé des points dont l'abscisse est comprise entre 100 et 250 et l'ordonnée comprise entre 50 et 200.



Un pixel de coordonnées (`x` ; `y`) appartient à ce carré si et seulement si `100 <= x && x <= 250 && 50 <= y && y <= 200`. On obtient donc le programme :

```
Isn.initDrawing("Carré rouge",10,10,400,400);
for (x = 0; x <= 399; x = x + 1) {
  for (y = 0; y <= 399; y = y + 1) {
    if (100 <= x && x <= 250 && 50 <= y && y <= 200) {
      Isn.drawPixel(x,y,255,0,0);}}}
```

Dans ce cas, il n'est cependant pas nécessaire de balayer toute la fenêtre graphique et un autre programme possible est :

```
Isn.initDrawing("Carré rouge",10,10,400,400);
for (x = 100; x <= 250; x = x + 1) {
  for (y = 50; y <= 200; y = y + 1) {
    Isn.drawPixel(x,y,255,0,0);}}
```

Exercice 19.2

Écrire un programme qui dessine ce même carré mais sans le remplir.

Exercice 19.3

Écrire un programme qui dessine un disque de centre (`a` ; `b`) et de rayon `r`.



Exercice 19.4

Tracer un segment pixel par pixel est facile quand ce segment est horizontal ou vertical, mais cela est un peu plus difficile quand il est oblique. Pour tracer un segment qui va du point (x ; y) au point (x' ; y') quand $|y' - y| \leq |x' - x|$, c'est-à-dire quand le segment est plutôt horizontal, on cherche à dessiner un pixel dans chaque colonne d'abscisse comprise entre x et x'.

1. Déterminer une équation cartésienne de la droite passant par les points de coordonnées (x ; y) et (x' ; y').
2. En supposant que $x \leq x'$, montrer que le point de la droite d'abscisse X a pour ordonnée $y + (X - x)(y' - y) / (x' - x)$. Dans la pratique, pour dessiner effectivement le pixel, ce nombre sera arrondi à l'entier le plus proche.
3. Écrire un programme qui trace ainsi un segment de type « plutôt horizontal ». On prendra soin de prévoir le cas où $x' \leq x$.
4. De même, quand le segment est plutôt vertical, c'est-à-dire quand $|x' - x| \leq |y' - y|$, on cherche à dessiner un pixel dans chaque ligne d'ordonnée comprise entre y et y'. Déterminer l'abscisse du point de la droite dont l'ordonnée est y et compléter l'algorithme pour le cas des segments plutôt verticaux.
5. Rechercher sur le Web ce qu'est *l'algorithme de Bresenham* et comment il améliore celui que l'on vient de construire.



Exercice 19.5

Tracer un cercle de centre (`a` ; `b`) et de rayon `r` en balayant les abscisses `x` de `a - r` à `a + r` et en calculant les valeurs de `y` à partir de l'équation $(x - a)^2 + (y - b)^2 = r^2$. Même question en balayant les ordonnées. Comment éviter les discontinuités ?



Exercice 19.6

Tracer, pour t variant de 0 à 10 000, la courbe définie par $x(t) = 256 + 250 \cos(0{,}0015\ t)$ et $y(t) = 256 + 250 \sin(kt)$, avec $k = 0{,}0045$, ceci dans une fenêtre de 512 pixels sur 512 pixels. C'est une *courbe de Lissajous*. Faire varier k de 0,0015 à 0,0090 et explorer les différentes courbes obtenues.

D'autres instructions dessinent des segments, des cercles et des disques, sans avoir à le faire pixel par pixel.

L'instruction `Isn.drawLine(x1,y1,x2,y2,rouge,vert,bleu);` trace un segment, de couleur `rouge`, `vert`, `bleu`, qui va du point (`x1` ; `y1`) au point (`x2` ; `y2`).

L'instruction `Isn.drawCircle(x,y,rho,rouge,vert,bleu);` trace un cercle, de couleur `rouge`, `vert`, `bleu`, de centre (`x` ; `y`) et de rayon `rho`.

L'instruction `Isn.paintCircle(x,y,rho,rouge,vert,bleu);` trace un disque, de couleur `rouge`, `vert`, `bleu`, de centre (`x` ; `y`) et de rayon `rho`.

Dessiner en trois dimensions

Les méthodes que l'on utilise aujourd'hui pour dessiner des images en trois dimensions remontent à la Renaissance quand, bien avant que les ordinateurs existent, les peintres ont commencé à mettre au point différentes méthodes de représentation de l'espace en *perspective* et à les utiliser dans leurs tableaux. Ces peintres sont finalement arrivés à une conclusion qui, en langage moderne, s'exprime assez simplement : un point de l'espace de coordonnées $(x ; y ; z)$ où l'axe des x va de gauche à droite, l'axe des y de bas en haut et celui des z de proche à loin, doit être représenté sur un tableau par un point de coordonnées $X = x / z$ et $Y = y / z$. Par exemple, le point de coordonnées (0 ; 1 ; 2) est représenté sur le tableau par le point de coordonnées (0 ; 1/2). Dans ce système de représentation, la coordonnée z doit toujours être strictement positive : les points dont la coordonnée z est négative correspondent aux points de l'espace qui sont dans le dos du peintre et qu'il ne représente donc pas.

Les points de coordonnées (0 ; 0 ; 1) et (0 ; 1 ; 1), qui sont à une distance 1 dans l'espace, sont représentés par deux points (0 ; 0) et (0 ; 1), qui sont aussi à une distance 1 sur le tableau. En revanche, les points de coordonnées (0 ; 0 ; 2) et (0 ; 1 ; 2) qui sont aussi à une distance 1 dans l'espace, sont représentés par deux points (0 ; 0) et (0 ; 1/2), qui sont à une distance 1/2 sur le tableau : plus un objet est loin, plus sa représentation sur le tableau est petite.

ALLER PLUS LOIN **Taille du tableau**

La taille du tableau est déterminée par la partie de l'espace que l'on veut représenter. Si l'on décide, par exemple, que x et y varient entre -1 et 1 et z entre 1 et l'infini, alors les coordonnées sur le tableau varient entre -1 et 1 : un objet peut être à l'extérieur du tableau parce qu'il est trop à gauche, trop à droite, trop en haut, trop en bas ou trop près, mais pas parce qu'il est trop loin. Cela est souvent rappelé dans les tableaux de la Renaissance : quand le peintre représente une scène qui se passe dans une pièce, il laisse souvent une porte ou une fenêtre ouverte sur une petite scène beaucoup plus loin. C'est, par exemple, le cas de cette *Annonciation* de Botticelli.



Si l'on veut maintenant représenter un tableau où les points sont repérés par des coordonnées $(X ; Y)$ qui varient entre -1 et 1 dans une fenêtre

graphique où les points sont repérés par des coordonnées (i ; j) qui varient entre 0 et 399, il faut faire un changement de repère et représenter le point du tableau de coordonnées (X ; Y) par le pixel de coordonnées $i = 200 + 200\ X$ et $j = 200 - 200\ Y$. Le signe - est dû au fait que l'axe vertical est orienté vers le haut dans le tableau et vers le bas dans la fenêtre graphique. Au bout du compte, le point de l'espace de coordonnées (x ; y ; z) est représenté par le pixel de coordonnées $i = 200 + 200\ x / z$ et $j = 200 - 200\ y / z$.

Par exemple, on dessine le cube dont une face $ABCD$ est dans le plan $z = 2$ et une autre face $A'B'C'D'$ est plus loin, dans le plan $z = 4$:

- A = (-1 ; -1 ; 2), B = (-1 ; 1 ; 2), C = (1 ; 1 ; 2), D = (1 ; -1 ; 2) ;
- A' = (-1 ; -1 ; 4), B' = (-1 ; 1 ; 4), C' = (1 ; 1 ; 4), D' = (1 ; -1 ; 4).

On commence par calculer les coordonnées des pixels représentant chacun de ces points en utilisant les formules $i = 200 + 200\ x / z$ et $j = 200 - 200\ y / z$:

- *A : (100 ; 300), B : (100 ; 100), C : (300 ; 100), D : (300 ; 300) ;*
- A' : (150 ; 250), B' : (150 ; 150), C' : (250 ; 150), D' : (250 ; 250).

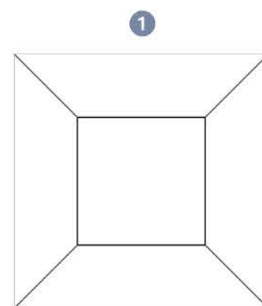
Il ne reste plus qu'à tracer les quatre segments de la première face ([AB], [BC], [CD] et [DA]), les quatre de la seconde ([$A'B'$], [$B'C'$], [$C'D'$] et [$D'A'$]) et les quatre qui relient chaque sommet d'une face au sommet homologue de l'autre ([AA'], [BB'], [CC'], [DD']).

```
// Face avant
Isn.drawLine(100,300,100,100,0,0,0);
Isn.drawLine(100,100,300,100,0,0,0);
Isn.drawLine(300,100,300,300,0,0,0);
Isn.drawLine(300,300,100,300,0,0,0);

// Face arrière
Isn.drawLine(150,250,150,150,0,0,0);
Isn.drawLine(150,150,250,150,0,0,0);
Isn.drawLine(250,150,250,250,0,0,0);
Isn.drawLine(250,250,150,250,0,0,0);

// Arêtes fuyantes
Isn.drawLine(100,300,150,250,0,0,0);
Isn.drawLine(100,100,150,150,0,0,0);
Isn.drawLine(300,100,250,150,0,0,0);
Isn.drawLine(300,300,250,250,0,0,0);
```

1



On obtient alors l'image 1 où la face du cube la plus proche est représentée par le grand carré, la face la plus lointaine par le petit carré et les quatre autres par des trapèzes. Dans ce dessin, on suppose que la face antérieure du cube est transparente, si bien que l'on voit l'intérieur du cube. Si on l'avait supposée opaque, il aurait fallu ne pas dessiner la partie du dessin cachée par cette face.

On peut imaginer que ce cube représente une pièce : la face inférieure est le sol, la face supérieure est le plafond et les trois faces verticales, représentées par les deux trapèzes latéraux et le petit carré sont les murs. Le quatrième mur est supposé ouvert, ou transparent, ou derrière le peintre, afin que l'on puisse voir l'intérieur de la pièce.

On peint maintenant chacune de ces faces. On peut commencer par peindre les murs latéraux en ambre jaune (r = 240, v = 195, b = 0). Il faut pour cela colorier tous les pixels contenus à l'intérieur du trapèze : l'écriture des deux boucles imbriquées est alors un peu plus complexe que pour le carré rouge précédent, car toutes les colonnes du trapèze n'ont pas la même hauteur. Pour traduire cela, les bornes entre lesquelles l'ordonnée j des pixels à colorier varie dépendront de l'abscisse i : on retrouve ici les équations, $j = i$ et $j = 400 - i$, des droites qui représentent les arêtes fuyantes du cube.

```
for (i = 100; i <= 150; i = i + 1) {
  for (j = i; j <= 400 - i; j = j + 1) {
    Isn.drawPixel(i,j,240,195,0);}}

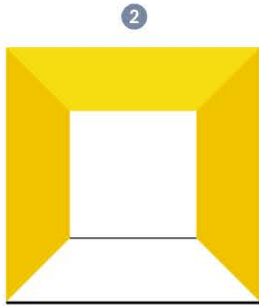
for (i = 250; i <= 300; i = i + 1) {
  for (j = 400 - i; j <= i; j = j + 1) {
    Isn.drawPixel(i,j,240,195,0);}}
```

On peint de même le plafond en jaune bouton d'or (r = 246, v = 220, b = 18). Cette fois, le trapèze a ses bases horizontales. C'est donc l'ordonnée j qui est choisie en premier dans la boucle externe, et les bornes de l'abscisse i qui dépendent de j :

```
for (j = 100; j <= 150; j = j + 1) {
  for (i = j; i <= 400 - j; i = i + 1) {
    Isn.drawPixel(i,j,246,220,18);}}
```

On obtient ainsi l'image 2.

2



On peut dessiner un carrelage sur le sol de la pièce que l'on vient de dessiner, c'est-à-dire la face inférieure du cube. Un point de coordonnées (x ; -1 ; z) de cette face inférieure est représenté par un pixel de la fenêtre graphique de coordonnées $i = 200 + 200\ x / z$ et $j = 200 + 200 / z$.

Réciproquement, le pixel de coordonnées (i ; j) représente le point de coordonnées $x = (i - 200) / (j - 200)$, $y = -1$, $z = 200 / (j - 200)$.

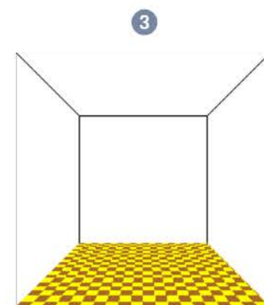
La coordonnée x varie entre -1 et 1 et la coordonnée z entre 2 et 4. Si on l'on découpe ce rectangle en 400 dalles carrées de 0,1 de coté, le point de coordonnées (i ; j) appartient à la dalle située dans la colonne $[10\ ((i - 200) / (j - 200) + 1)]$, où $[x]$ est la partie entière de x, c'est-à-dire $[10\ (i + j - 400) / (j - 200)]$, et dans la ligne $[10\ (200 / (j - 200) - 2)]$, c'est-à-dire $[10\ (600 - 2\ j) / (j - 200)]$.

Remarque — Carrelages et perspective

Les peintres de la Renaissance peignaient souvent des carrelages, comme Botticelli dans son Annonciation, car les carrelages sont faciles à représenter et ils soulignent la perspective.

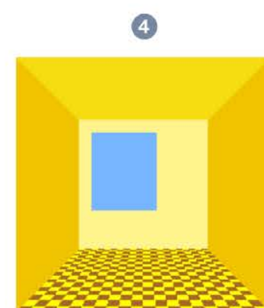
Pour construire un carrelage en damier, il suffit de choisir une couleur pour les dalles dont la somme des numéros de ligne et de colonne est paire et une autre pour celles dont la somme est impaire. Il faut donc dessiner le pixel $(i\,;j)$ d'une couleur si $[10\,(i + j - 400) / (j - 200)] + [10\,(600 - 2\,j) / (j - 200)]$ est pair et d'une autre couleur quand ce nombre est impair :

```
for (j = 250; j <= 300; j = j + 1) {
  for (i = 400 - j; i <= j; i = i + 1) {
    if ((10 * (i + j - 400) / (j - 200) + 10 * (600 - 2*j)
          / (j - 200))%2 == 0) {
      Isn.drawPixel(i,j,167,103,38);}
    else {
      Isn.drawPixel(i,j,255,255,0);}}}
```


3

En alternant des dalles alezan ($r = 167$, $v = 103$, $b = 38$) et jaunes ($r = 255$, $v = 255$, $b = 0$), on obtient l'image 3. Il ne reste plus qu'à peindre le mur du fond en beurre frais ($r = 255$, $v = 244$, $b = 141$), en laissant, comme les peintres de la Renaissance, une fenêtre ouverte sur le ciel bleu ciel ($r = 119$, $v = 181$, $b = 254$) :

```
for (i = 150; i <= 250; i = i + 1) {
  for (j = 150; j <= 250; j = j + 1) {
    if (160 <= i && i <= 210 && 160 <= j && j <= 220) {
      Isn.drawPixel(i,j,119,181,254);}
    else {
      Isn.drawPixel(i,j,255,244,141);}}}
```


4

pour terminer l'image et obtenir 4.

Exercice 19.7

On considère un repère $(O; i; j; k)$ tel que l'œil du peintre soit en O et le tableau soit dans le plan $z = 1$. Soit un point A de coordonnées $(x\,;y\,;z)$. Trouver une représentation paramétrique de la droite (AO) support du rayon lumineux qui va du point A à l'œil du peintre. On représente ce point A sur le tableau par le point A' intersection de cette droite avec le plan du tableau $z = 1$. Montrer que les coordonnées du point A' sont $(x / z\,, y / z\,, 1)$.

Produire un fichier au format PPM

Si au lieu de dessiner cette image dans une fenêtre graphique, on veut l'enregistrer dans un fichier, par exemple au format PPM (voir le chapitre 9), afin de pouvoir l'inclure dans une page web ou l'attacher à un

courrier, on doit d'abord représenter cette image dans un tableau, au sens que l'on a donné à ce mot au chapitre 3, puis produire un fichier au format PPM à partir de ce tableau. Pour représenter une image en niveaux de gris, il suffit d'utiliser un tableau bidimensionnel `t`, dont la case `t[i][j]` contient la valeur du pixel de coordonnées (`i` ; `j`). Pour représenter une image en couleurs, on utilise trois tableaux bidimensionnels `rouge`, `vert` et `bleu`, les cases `rouge[i][j]`, `vert[i][j]` et `bleu[i][j]` contenant les trois composantes de la couleur du pixel de coordonnées (`i` ; `j`).

On peut alors transformer le programme précédent en remplaçant toutes les instructions `Isn.drawPixel(i,j,r,v,b);` par

```
rouge[i][j] = r;
vert[i][j] = v;
bleu[i][j] = b;
```

Par exemple, le dessin du carrelage s'écrit désormais :

```
for (j = 250; j <= 300; j = j + 1) {
  for (i = 400 - j; i <= j; i = i + 1) {
    if ((10 * (i + j - 400) / (j - 200) + 10 * (600 - 2*j)
        / (j - 200))%2 == 0) {
      rouge[i][j] = 167;
      vert[i][j] = 103;
      bleu[i][j] = 38;}
    else {
      rouge[i][j] = 255;
      vert[i][j] = 255;
      bleu[i][j] = 0;}}}
```

Un fichier PPM

```
P3
#
400
400
255
rouge[0][0]
vert[0][0]
bleu[0][0]
rouge[1][0]
vert[1][0]
bleu[1][0]
...
```

Une fois le dessin terminé, on peut l'enregistrer au format PPM, qui est un fichier texte de la forme ci-contre.

La première ligne contient les caractères `P3` pour indiquer que c'est un fichier au format PPM, la deuxième est un commentaire, la troisième la largeur de l'image, la quatrième sa hauteur, la cinquième la valeur `255` pour indiquer que les valeurs des pixels vont de 0 à 255, puis sur les lignes suivantes les trois valeurs rouge, vert et bleu en énumérant les pixels de gauche à droite et de haut en bas. Le programme qui crée un tel fichier s'écrit :

```
fichier = Isn.openOut("botticelli.ppm");
Isn.printlnToFile(fichier,"P3");
Isn.printlnToFile(fichier,"#");
Isn.printlnToFile(fichier,400);
Isn.printlnToFile(fichier,400);
Isn.printlnToFile(fichier,255);
for (j = 0; j <= 399; j = j + 1) {
  for (i = 0; i <= 399; i = i + 1) {
    Isn.printlnToFile(fichier,rouge[i][j]);
```

```
    Isn.printlnToFile(fichier,vert[i][j]);
    Isn.printlnToFile(fichier,bleu[i][j]);}}
Isn.closeOut(fichier);
```

Lire un fichier au format PPM

Inversement, on peut écrire un programme qui lit un fichier au format PGM ou PPM dans un tableau ou dans trois, selon que l'image est en niveaux de gris ou en couleurs.

La seule difficulté, pour lire un tel fichier, est due au fait qu'il est possible d'insérer des commentaires dans un fichier au format PGM ou PPM, c'est-à-dire des lignes qui commencent par le caractère `#` et qui doivent être ignorées. En pratique cependant, les fichiers au format PGM et PPM n'ont généralement qu'un seul commentaire, à la deuxième ligne.

```
P2
# une photo prise au Louvre
181
279
255
86
94
103
…
```

On se limite donc à des fichiers de cette forme si bien que les fichiers PGM que l'on lit sont de la forme suivante :

- deux lignes qui peuvent être ignorées,
- la largeur de l'image, suivie d'un retour à la ligne ou d'un espace,
- la hauteur de l'image, suivie d'un retour à la ligne ou d'un espace,
- la valeur maximale utilisée pour exprimer les niveaux de gris, suivie d'un retour à la ligne ou d'un espace,
- la liste des pixels, ligne par ligne, de haut en bas et de gauche à droite, séparés par des retours à la ligne ou des espaces.

On peut lire un tel fichier avec le programme suivant :

```
fichier = Isn.openIn("maison.pgm");
s = Isn.readStringFromFile(fichier);
s = Isn.readStringFromFile(fichier);
largeur = Isn.readIntFromFile(fichier);
hauteur = Isn.readIntFromFile(fichier);
max = Isn.readIntFromFile(fichier);
gris = new int [largeur][hauteur];
for (j = 0; j <= hauteur - 1; j = j + 1) {
  for (i = 0; i <= largeur - 1; i = i + 1) {
    gris[i][j] = Isn.readIntFromFile(fichier);}}
Isn.closeIn(fichier);
```

et ensuite afficher cette image dans une fenêtre :

```
Isn.initDrawing("Pgm",10,10,largeur,hauteur);
for (i = 0; i <= largeur - 1; i = i + 1) {
  for (j = 0; j <= hauteur - 1; j = j + 1) {
    valeurgris = gris[i][j] * 255 / max;
    Isn.drawPixel(i,j,valeurgris,valeurgris,valeurgris);}}
```

La lecture et l'affichage d'une image en couleurs, au format PPM sont similaires, sauf qu'il faut lire trois nombres pour chaque pixel et les stocker dans trois tableaux : `rouge`, `vert` et `bleu`.

Transformer les images



Une fois une image représentée dans un tableau, il est facile de la transformer. Par exemple, on peut inverser la quantité de chaque couleur :

```
for (j = 0; j <= hauteur - 1; j = j + 1) {
  for (i = 0; i <= largeur - 1; i = i + 1) {
    rougebis[i][j] = max - rouge[i][j];
    vertbis[i][j] = max - vert[i][j];
    bleubis[i][j] = max - bleu[i][j];}}
```

ce qui transforme l'image ❶ en l'image ❷.

SAVOIR-FAIRE Transformer une image en couleurs en une image en niveaux de gris

On remplace chaque pixel de couleur r, v, b par un pixel dont le niveau de gris est la moyenne des nombres r, v et b.

Exercice 19.8 (avec corrigé)

Écrire un programme qui transforme une image en couleurs en une image en niveaux de gris.



```
for (i = 0; i <= largeur - 1; i = i + 1) {
  for (j = 0; j <= hauteur - 1; j = j + 1) {
    gris[i][j] = (rouge[i][j] + bleu[i][j] + vert[i][j])/3;}}
```

Exercice 19.9

Écrire un programme qui transforme une image en couleurs en une image en niveaux de gris, non pas en faisant la moyenne, mais en gardant un seul des trois nombres *r*, *v* et *b* et en ignorant les autres. Comparer le résultat obtenu avec le résultat de l'exercice 19.8.

SAVOIR-FAIRE Augmenter le contraste d'une image en niveaux de gris

On fixe un seuil et on remplace tous les pixels plus clairs que ce seuil par un pixel blanc et tous les pixels plus sombres que ce seuil par un pixel noir.

Exercice 19.10 (avec corrigé)

Écrire un programme qui augmente le contraste d'une image en se fixant comme seuil la valeur `max` / 5.



```
for (i = 0; i <= largeur - 1; i = i + 1) {
  for (j = 0; j <= hauteur - 1; j = j + 1) {
    if (gris[i][j] <= max/5) {
      grisbis[i][j] = 0;}
    else {
      grisbis[i][j] = max;}}}
```

Exercice 19.11

Écrire un programme qui augmente le contraste d'une image en se fixant comme seuil la valeur 4 `max` / 5. Ce programme est-il bien adapté pour les images claires ou les images sombres ?

Exercice 19.12

À partir d'une image en niveaux de gris, produire les images suivantes.



SAVOIR-FAIRE Modifier la luminance d'une image

On ajoute ou on retranche une constante à la valeur de chacun des pixels.

Exercice 19.13 (avec corrigé)

Écrire un programme qui ajoute max / 4 à tous les pixels d'une image en niveaux de gris, en remplaçant les valeurs qui dépassent la valeur maximale par la valeur maximale elle-même.

```
diff = max / 4;
for (x = 0; x <= largeur - 1; x = x + 1) {
  for (y = 0; y <= hauteur - 1; y = y + 1) {
    grisbis[x][y] = Math.min(gris[x][y] + diff,max);}}
```

Exercice 19.14

Écrire un programme qui augmente la luminance d'une image en couleurs.



Exercice 19.15

Écrire un programme qui multiplie toutes les valeurs des pixels de l'image par un nombre g positif en remplaçant les valeurs qui dépassent la valeur maximale par la valeur maximale elle-même. Ce programme modifie-t-il la luminance ou le contraste de l'image ? Que se passe-t-il quand le coefficient multiplicateur g est très petit ? Et quand il est très grand ? Au lieu de transformer la valeur v en g * v, la transformer en 128 + g * (v - 128), toujours en gardant la même valeur maximale et en remplaçant les valeurs qui la dépassent par la valeur maximale elle-même, et celles qui dépassent 0 par 0. Que se passe-t-il quand g est très grand ?



Exercice 19.16

Prendre une photo sombre, par exemple avec un téléphone, et utiliser la transformation qui à v associe 255.0 * Math.pow(v / 255.0, gamma) avec 0 < gamma < 1. Que se passe-t-il ? À quoi peut servir cette transformation ?

SAVOIR-FAIRE Changer la taille d'une image

On calcule la nouvelle image pixel par pixel.

Exercice 19.17 (avec corrigé)

Écrire un programme qui double la taille d'une image en niveaux de gris, en remplaçant chaque pixel par un carré de deux pixels sur deux pixels du même niveau de gris.

```
largeurbis = 2*largeur;
hauteurbis = 2*hauteur;
grisbis = new int [largeurbis][hauteurbis];
for (i = 0; i <= largeurbis - 1; i = i + 1) {
  for (j = 0; j <= hauteurbis - 1; j = j + 1) {
    grisbis[i][j] = gris[i/2][j/2];}}
```

Exercice 19.18

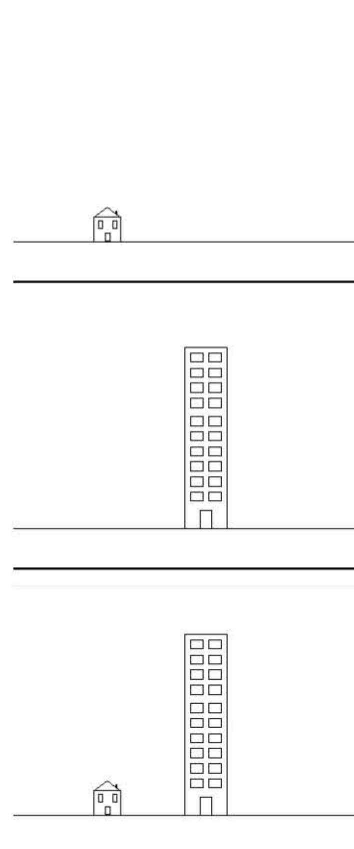
Écrire un programme qui divise la taille d'une image par deux, en remplaçant chaque carré de deux pixels sur deux pixels par un pixel dont la couleur est la moyenne de celles des quatre pixels qu'il remplace.

SAVOIR-FAIRE Fusionner deux images

On calcule la nouvelle image pixel par pixel, la valeur de chaque pixel étant le maximum des valeurs des pixels correspondant dans chacune des images à fusionner.

Exercice 19.19 (avec corrigé)

Écrire un programme qui fusionne deux images en niveaux de gris. Attention au cas où les images n'ont pas la même taille ou n'utilisent pas la même valeur maximale pour exprimer les niveaux de gris.



```
// Calcul des dimensions maximales des deux images
if (largeur1 >= largeur2) {
  largeur3 = largeur1;}
else {
  largeur3 = largeur2;}
if (hauteur1 >= hauteur2) {
  hauteur3 = hauteur1;}
else {
  hauteur3 = hauteur2;}

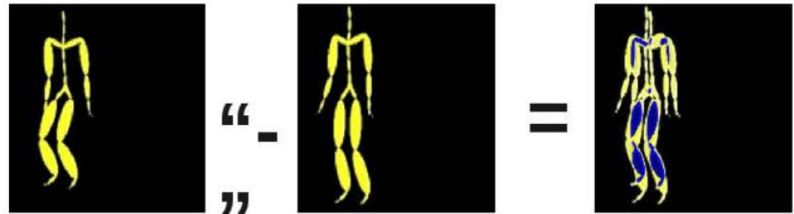
// Calcul du niveau de gris maximal
if (max1 >= max2) {
  max3 = max1;}
else {
  max3 = max2;}

// Calcul de l'image fusionnée
gris3 = new int [largeur3][hauteur3];
for (j = 0; j <= hauteur3 - 1; j = j + 1) {
  for (i = 0; i <= largeur3 - 1; i = i + 1) {
    // Si un pixel est en dehors d'une image on lui affecte
    // la valeur maximale
    if (i < largeur1 && j < hauteur1) {
      valeur1 = max3 * gris1[i][j] / max1;}
    else {
      valeur1 = max3;}
    if (i < largeur2 && j < hauteur2) {
      valeur2 = max3 * gris2[i][j] / max2;}
    else {
      valeur2 = max3;}
    if (valeur1 < valeur2) {
      gris3[i][j] = valeur1;}
    else {
      gris3[i][j] = valeur2;}}}
```



Exercice 19.20

À quoi peut correspondre la soustraction de deux images en niveaux de gris, c'est-à-dire l'image obtenue en soustrayant à la valeur de chaque pixel de la première image la valeur du pixel correspondant de la seconde ? On s'inspirera du schéma ci-après. Prendre deux photos d'un objet sur une table en bougeant un peu l'objet entre les deux photos et calculer la valeur absolue de la différence pixel à pixel. Quelle peut être une application de cet algorithme ?



SAVOIR-FAIRE Lisser une image pour éliminer ses petits défauts et en garder les grands traits

La valeur d'un pixel de la nouvelle image est la moyenne des valeurs de ce pixel et des pixels qui l'entourent (à gauche, à droite, au-dessus et en-dessous).

Exercice 19.21 (avec corrigé)

Écrire un programme qui lisse une image. Attention au fait que les pixels sur le bord gauche de l'image n'ont pas de pixel à leur gauche, ceux du bord droit, pas de pixel à leur droite, etc.

```
for (x = 0; x <= largeur - 1; x = x + 1) {
  for (y = 0; y <= hauteur - 1; y = y + 1) {
    grisbis[x][y] = (gris[x][y]
      + gris[Math.max(x - 1,0)][y]
      + gris[Math.min(x + 1,largeur - 1)][y]
      + gris[x][Math.max(y - 1,0)]
      + gris[x][Math.min(y + 1,hauteur - 1)])/5;}}
```

Et si on répète l'opération plusieurs fois, l'image sera de plus en plus lisse mais aussi plus floue, comme illustré ici avec l'image initiale à gauche et l'image lissée à droite en appliquant l'opération précédente 20 fois.





Il existe des algorithmes plus complexes qui, pour une image couleur, font la moyenne sur les petites variations pour gommer les détails les moins importants, mais pas sur les plus grands traits, permettant ainsi de bien les faire ressortir, on obtient alors de meilleurs résultats.





Exercice 19.22

Considérer l'image qui suit avec des valeurs de pixels entre 0 et 9. Remplacer la valeur de chaque pixel par la moyenne des valeurs des pixels à sa gauche et à sa droite. On arrondit à la valeur entière la plus proche à chaque fois. On prendra 0 au bord de l'image.

Dessiner le résultat obtenu. On économisera beaucoup de calculs en tenant compte des symétries.

```
0000000000
0088888800
0880000880
0800000080
0800000080
0800000080
0800000080
0880000880
0088888800
0000000000
```



Exercice 19.23

Reprendre l'exercice précédent, mais en recommençant encore deux fois. Commenter le résultat obtenu. Que se passe-t-il si on recommence indéfiniment ?

Exercice 19.24

Écrire un programme qui extrait les contours d'une image `gris` en niveaux de gris. Pour cela, on fixe un seuil et on construit une image dont le pixel (`i` ; `j`) est noir si l'intensité varie fortement autour du pixel (`i` ; `j`) de `gris`, c'est-à-dire si `Math.abs(gris[i+1][j] - gris[i][j]) + Math.abs(gris[i][j+1] - gris[i][j])` est supérieur au seuil fixé, et blanc si ce n'est pas le cas.

Exercice 19.25

Dessiner une image d'au moins 200 pixels par 200 pixels, où la valeur de chaque pixel est aléatoire. Observer le résultat obtenu : n'y a-t-il pas quelques régularités ? Elles sont évidemment dues au hasard : il faut se méfier de surinterpréter un phénomène aléatoire dans lequel on croit apercevoir des régularités.

ALLER PLUS LOIN **Comment sont dessinées nos maisons ?**

Les algorithmes géométriques que l'on a vus dans ce chapitre sont, bien entendu, utilisés par l'industrie du dessin animé, mais ils sont également utilisés dans beaucoup d'autres secteurs de l'industrie, pour la conception assistée par ordinateur de voitures, de bateaux, d'avions, de bâtiments, etc.
Il y a encore quelques dizaines d'années, les architectes dessinaient à la main les plans des bâtiments qu'ils concevaient à la planche à dessin sur du papier calque. Effacer un trait demandait alors de gratter le papier avec une lame de rasoir : il valait donc mieux éviter de se tromper deux fois. Les planches à dessin ont été remplacées par des logiciels de conception assistée par ordinateur, qui permettent de dessiner des plans avec une précision beaucoup plus grande et surtout de les modifier *ad libitum*.
Ils ont par la suite été complétés par des logiciels de dessin en trois dimensions, qui représentent les bâtiments sous forme de *maquettes virtuelles*, beaucoup plus lisibles que des plans, surtout pour les non spécialistes.
Les ingénieurs dessinaient ensuite d'autres plans et faisaient des notes de calcul à la main ou en utilisant une calculatrice. Ils utilisent aujourd'hui, pour ces calculs, des logiciels fondés sur la méthode des éléments finis, qui décomposent les bâtiments en des milliers de petits morceaux et calculent les forces exercées sur chacun de ces morceaux.
Des évolutions récentes vont vers l'utilisation d'un format unique de description des bâtiments, utilisé à la fois par les architectes, les ingénieurs de conception et les ingénieurs qui organisent le travail sur les chantiers.
Ces transformations ont permis de concevoir et de calculer des bâtiments beaucoup plus complexes que par le passé, comme le musée Guggenheim de Bilbao. Ainsi, on comprend rétrospectivement que la raison pour laquelle beaucoup de bâtiments étaient, par le passé, de simples parallélépipèdes, est que les parallélépipèdes sont faciles à dessiner et à calculer.
Ces méthodes ont aussi permis une présentation plus lisible des bâtiments dans lesquels il est désormais possible de se promener avant même qu'ils ne soient construits. Enfin, elles favorisent une meilleure communication entre les différents corps de métier. Par exemple, quand un bâtiment demande la découpe d'une pièce sur mesure, le fichier de commande de la machine numérique qui découpe cette pièce peut être produit directement à partir des logiciels de conception assistée par ordinateur.

Ai-je bien compris ?

- De quelles instructions a-t-on besoin pour dessiner à l'écran ?
- Quel est le principe du dessin en trois dimensions ?
- Quelles sont les principales transformations d'une image ?

La dichotomie 20

CHAPITRE AVANCÉ



Donald Knuth (1938-) est un des fondateurs de l'algorithmique, la partie de l'informatique qui étudie les propriétés des algorithmes, indépendamment de leur expression dans un langage de programmation particulier. Le troisième volume de son livre *The art of computer programming*, intitulé *Sorting and searching* est entièrement consacré aux algorithmes de tri et de recherche en table. Pour écrire ce livre, il a d'abord écrit un logiciel de traitement de texte : TeX, dont le nom est formé des trois premières lettres du mot *technè*, qui signifie à la fois « art » et « technique ».

Ou diviser pour régner.

Dans ce chapitre, nous voyons une méthode algorithmique générale qui permet, entre autres choses, de trouver rapidement un mot dans un dictionnaire : ouvrir le dictionnaire au milieu et s'interroger : « la première lettre du mot cherché est-elle avant ou après ? ».

Cette méthode fonctionne plus généralement dès que nous cherchons à retrouver un élément dans un tableau ordonné selon une relation d'ordre total, notion que nous définissons. Cette méthode est rapide et s'applique à de nombreux problèmes : la recherche d'un élément dans une table ordonnée, la conversion d'une valeur analogique en une valeur numérique, la recherche d'un zéro d'une fonction continue et strictement monotone, etc.

La recherche en table

Au chapitre 3, nous avons écrit un programme qui gère un répertoire constitué de deux tableaux, contenant l'un des noms et l'autre des numéros de téléphone. Ce programme attend en entrée un nom et indique le numéro de téléphone correspondant, ou bien indique que ce nom n'appartient pas au répertoire.

Ce problème est un exemple d'un problème important en algorithmique : la *recherche en table*. En effet, beaucoup de systèmes informatiques, tels les dictionnaires, les moteurs de recherche, les systèmes d'information des banques et des administrations, servent essentiellement, comme ce répertoire, à stocker des informations et à les restituer quand on les interroge. Le programme que nous avons écrit au chapitre 3 est le suivant :

```
s = Isn.readString();
i = 0;
while (i < 10 && !Isn.stringEqual(s,nom[i])) {
  i = i + 1;}
if (i < 10) {
  System.out.println(tel[i]);}
else {
  System.out.println("Inconnu");}
```

Ce programme recherche, dans le tableau `nom`, l'indice de la chaîne `s` entrée par l'utilisateur, en comparant cette chaîne successivement à tous les éléments du tableau. Il suffit ensuite d'afficher l'élément de même indice du tableau `tel`. On peut instrumenter ce programme en ajoutant une instruction `System.out.print(".");` dans la boucle afin de visualiser le nombre de comparaisons effectuées.

```
s = Isn.readString();
i = 0;
while (i < 10 && !Isn.stringEqual(s,nom[i])) {
  System.out.print(".");
  i = i + 1;}
System.out.println(".");
if (i < 10) {
  System.out.println(tel[i]);}
else {
  System.out.println("Inconnu");}
```

On obtient alors trois points `...` quand on cherche le numéro de téléphone de Charles, qui est plutôt au début du tableau, mais dix points `..........` quand on cherche celui de Jérôme, qui est à la fin.

Si on utilisait cette méthode pour chercher un mot dans un dictionnaire, on ouvrirait celui-ci à la première page et on comparerait le mot recherché au premier mot du dictionnaire, puis au deuxième, puis au

troisième, etc. jusqu'à trouver le mot recherché, ou arriver au dernier mot du dictionnaire. Un dictionnaire courant contenant 60 000 mots et une comparaison prenant une demi-seconde, il faudrait, dans le pire des cas, 30 000 secondes, soit huit heures et vingt minutes, pour trouver le mot recherché ou se convaincre qu'il n'appartient pas au dictionnaire.

Bien entendu, ce n'est pas ainsi que l'on procède : on ouvre le dictionnaire au milieu, on compare le mot recherché au mot médian. Si le mot recherché est avant le mot médian dans l'ordre alphabétique, on élimine la seconde moitié du dictionnaire, sans même la regarder ; s'il est après le mot médian, on élimine la première moitié. En recommençant avec le demi-dictionnaire restant, on élimine ensuite un demi-demi-dictionnaire et on continue jusqu'à trouver le mot en question ou obtenir l'ensemble vide, auquel cas le mot recherché n'est pas dans le dictionnaire. Cette algorithme de recherche d'un élément dans une table s'appelle la recherche par *dichotomie* (*tomia,* couper, *dikha,* en deux).

Si on cherche un terme dans un dictionnaire de 60 000 mots :

- Après 1 comparaison, on le cherche dans un ensemble d'au plus 30 000 mots.
- Après 2 comparaisons, on le cherche dans un ensemble d'au plus 15 000 mots.
- Après 3 comparaisons, on le cherche dans un ensemble d'au plus 7 500 mots.
- Après 4 comparaisons, on le cherche dans un ensemble d'au plus 3 750 mots.
- Après 5 comparaisons, on le cherche dans un ensemble d'au plus 1 875 mots.
- Après 6 comparaisons, on le cherche dans un ensemble d'au plus 937 mots.
- Après 7 comparaisons, on le cherche dans un ensemble d'au plus 468 mots.
- Après 8 comparaisons, on le cherche dans un ensemble d'au plus 234 mots.
- Après 9 comparaisons, on le cherche dans un ensemble d'au plus 117 mots.
- Après 10 comparaisons, on le cherche dans un ensemble d'au plus 58 mots.
- Après 11 comparaisons, on le cherche dans un ensemble d'au plus 29 mots.
- Après 12 comparaisons, on le cherche dans un ensemble d'au plus 14 mots.
- Après 13 comparaisons, on le cherche dans un ensemble d'au plus 7 mots.
- Après 14 comparaisons, on le cherche dans un ensemble d'au plus 3 mots.
- Après 15 comparaisons, on le cherche dans un ensemble d'au plus 1 mot.

Au bout de 16 comparaisons seulement, on a trouvé le mot dans le dictionnaire, ou on sait qu'il n'y est pas : 8 secondes suffisent donc, contre 8 heures et 20 minutes, soit un temps de recherche divisé par 3 750.

Au cours d'une recherche par dichotomie, soit on tombe sur le résultat, soit on divise par deux la taille de l'ensemble dans lequel on recherche le mot, ceci à chaque comparaison. De ce fait, le nombre de comparaisons nécessaires pour trouver un élément dans une table est, dans le pire des cas, le logarithme entier de la taille de la table.

Logarithme entier

On rappelle que le *logarithme entier* elog(x) d'un nombre x supérieur ou égal à 1 est le nombre de fois qu'il faut le diviser par deux pour obtenir un nombre inférieur ou égal à 1.

Le gain qui consiste à passer d'un nombre de comparaisons proportionnel au nombre d'éléments à un nombre de comparaison proportionnel à son logarithme entier est immense. Quand le nombre d'éléments atteint quelques millions ou quelques milliards, son logarithme entier ne vaut que 20 ou 30 :

n	1	2	4	8	16	..	256	..	1024	..	Million	..	Milliard
Logarithme entier de *n*	0	1	2	3	4	..	8	..	10	..	20	..	30

La dichotomie permet, par exemple, de rechercher en un maximum de 26 étapes, le nom de quelqu'un dans l'annuaire des 60 millions de Français.

Bien entendu, cet algorithme de recherche par dichotomie ne fonctionne que parce que, dans un dictionnaire, les mots sont ordonnés par ordre alphabétique. Si les mots avaient été dans le désordre, il aurait fallu tout fouiller. De manière générale, il faut que l'on ait défini, sur le type des éléments de la table, une relation d'ordre total, c'est-à-dire une relation ≤ qui soit :

- *réflexive* : un élément x est avant lui-même, au sens large $x \leq x$
- *anti-symétrique :* si x est avant y alors y n'est pas avant x, sauf s'ils sont égaux si $x \leq y$ et $y \leq x$ alors $x = y$
- *transitive* : si x est avant y et y avant z, alors x est avant z si $x \leq y$ et $y \leq z$ alors $x \leq z$
- et *totale* : de deux éléments, l'un est toujours avant l'autre, $x \leq y$ ou $y \leq x$.

La relation d'ordre habituelle sur les nombres entiers et l'ordre alphabétique sur les chaînes de caractères sont deux exemples de relations d'ordre totales.

Il faut, par ailleurs, que la table soit *ordonnée* relativement à cette relation, c'est-à-dire que si x est avant y dans la table, alors $x \leq y$.

On peut alors écrire un programme qui exprime l'algorithme de recherche par dichotomie. Deux variables `i` et `j` définissent l'intervalle du tableau `nom`, auquel l'indice de la chaîne de caractères `s` recherchée appartient, si jamais cette chaîne est dans la table. Tant que cet intervalle contient au moins deux éléments, c'est-à-dire tant que `i < j`, on calcule l'élément médian `k` de l'intervalle et on compare les chaînes de caractères `s` et `nom[k]`. Si ces deux chaînes de caractères sont identiques, on réduit l'intervalle au singleton [`k`, `k`], de manière à provoquer la fin du calcul. Si `s` est avant `nom[k]` dans l'ordre alphabétique, on réduit l'intervalle à [`i`, `k - 1`] et si `s` est après `nom[k]` dans l'ordre alphabétique, on réduit l'intervalle à [`k + 1`, `j`].

On s'arrête quand l'intervalle contient moins de deux éléments. Comme on va le voir, l'intervalle [`i`, `j`] est alors ou bien de la forme [`i`, `i`], qui est un singleton, ou bien de la forme [`i`,`i - 1`], qui est vide puisque `i` est plus grand que `i - 1`. Et `i` est toujours compris entre 0 et 9. Si la chaîne de

caractères `nom[i]` est identique à `s`, on a trouvé l'indice de la chaîne `s` dans le tableau `nom` ; si ce n'est pas le cas, la chaîne `s` n'est pas dans la table.

```
s = Isn.readString();
i = 0;
j = 9;
while (i < j) {
  k = (i + j) / 2;
  if (Isn.stringEqual(s,nom[k])) {
    i = k;
    j = k;}
  else {
    if (Isn.stringAlph(s,nom[k])) {
      j = k-1;}
    else {
      i = k+1;}}}
if (Isn.stringEqual(s,nom[i])) {
  System.out.println(tel[i]);}
else {
  System.out.println("Inconnu");}
```

On doit se convaincre de deux choses : d'une part que l'algorithme se termine après un certain nombre d'itérations, d'autre part que la réponse donnée par l'algorithme est correcte.

Le fait que la boucle se termine est dû au fait que le nombre `j - i + 1` d'éléments dans l'intervalle [`i`, `j`] est au moins divisé par deux à chaque itération. Après un nombre d'itérations inférieur au logarithme entier du nombre d'éléments dans la table, ce nombre est inférieur ou égal à 1 et la boucle se termine. En instrumentant le programme précédent pour compter le nombre de comparaisons, on s'aperçoit que ce nombre est toujours inférieur ou égal à 4, qui est le logarithme entier de 10.

Ensuite, pour démontrer que la réponse donnée par l'algorithme est correcte, on commence par montrer que si la chaîne de caractères `s` est dans la table, alors son indice appartient toujours à l'intervalle [`i`, `j`]. Cette propriété est un *invariant* de la boucle, c'est-à-dire une propriété qui reste vraie à chaque exécution du corps de la boucle. Ici, quand on réduit l'intervalle [`i`, `j`] à l'intervalle [`i`, `k - 1`] par exemple, c'est parce que l'on sait que la chaîne `s` est avant la chaîne `nom[k]` dans l'ordre alphabétique et donc que l'indice de la chaîne `s`, s'il existe, n'est pas dans l'intervalle [`k`, `j`]. La propriété reste donc vraie jusqu'à la fin de l'exécution de la boucle.

Enfin, on montre que quand on sort de la boucle, l'intervalle [`i`, `j`] est soit le singleton [`i`, `i`], soit l'intervalle vide [`i`, `i - 1`]. Dans les deux cas, `i` est compris entre les valeurs minimale et maximale de départ, ici 0 et 9. Pour cela, on montre un autre invariant de la boucle : si l'intervalle [`i`, `j`] n'est pas vide, alors ses bornes `i` et `j` sont comprises entre les valeurs minimale et maximale de départ, et s'il est vide, alors sa borne inférieure `i` est comprise entre les valeurs minimale et maximale de départ.

> **ALLER PLUS LOIN Ajouter un élément en temps logarithmique**
>
> Cet algorithme de recherche par dichotomie permet donc la recherche rapide d'un élément dans une table, puisque le nombre de comparaisons, et par conséquent le temps d'exécution du programme, est proportionnel au logarithme de la taille de la table et non à la taille de la table elle-même. En revanche, ajouter ou supprimer un élément de la table demande un temps de calcul proportionnel à la taille de la table, et non à son logarithme, puisque quand on ajoute ou supprime un élément au début du tableau, il faut décaler tous les autres éléments. Ce n'est pas très grave quand la table change peu, comme un dictionnaire, mais cela peut devenir un problème si elle change souvent. C'est pour cela qu'il existe d'autres manières, plus complexes, de programmer la recherche en table, qui rendent logarithmiques aussi bien la recherche, que l'ajout et la suppression d'un élément de la table. Bien que plus complexes, ces méthodes sont fondées sur les mêmes idées que celles que nous avons vues dans cet exemple.

- Si l'intervalle [`i`, `j`] contient au moins trois points, c'est-à-dire si `i` + 2 ≤ `j`, il n'est pas difficile de montrer que les nombres `k` - 1 et `k` + 1, où `k` = (`i` + `j`) / 2, sont tous les deux compris entre `i` et `j` au sens large. Le nouvel intervalle [`k`, `k`], [`i`, `k` - 1] ou [`k` + 1, `j`] est contenu dans [`i`, `j`] et donc ses bornes sont comprises entre les valeurs minimale et maximale de départ.
- Si l'intervalle [`i`, `j`] contient deux points, c'est-à-dire si `j` = `i` + 1, alors `k` = (`i` + `j`) / 2 est égal à `i`. Le nombre `k` + 1 est égal à `j` : il est compris entre `i` et `j` au sens large. En revanche, le nombre `k` - 1 est égal à `i` - 1. Dans ce cas, le nouvel intervalle est [`i`, `i`] ou [`j`, `j`] dont les bornes sont comprises entre les valeurs minimale et maximale de départ, ou l'intervalle vide [`i`, `i` - 1] dont la borne inférieure `i` est comprise entre les valeurs minimale et maximale de départ.

On sort de la boucle quand l'intervalle [`i`, `j`] contient zéro ou un point. Dans un cas comme dans l'autre, l'indice `i` est compris entre les valeurs minimale et maximale de départ. Si la chaîne de caractères `nom[i]` est identique à `s`, on a trouvé l'indice de la chaîne `s` dans le tableau `nom` ; si ce n'est pas le cas, la chaîne `s` n'est pas dans la table.

Exercice 20.1

On suppose que l'on a un annuaire qui contient les sept milliards d'êtres humains dans l'ordre alphabétique de leurs nom, prénom, lieu de naissance et date de naissance. Combien de comparaisons sont nécessaires pour retrouver une personne dans cet annuaire ?



Exercice 20.2

Programmer l'algorithme de recherche en table par dichotomie de façon récursive.

Exercice 20.3

Écrire un programme qui joue au jeu du « plus petit - plus grand » c'est-à-dire qui propose à un utilisateur de deviner un nombre entre 0 et 100 en lui indiquant à chaque tentative si le nombre proposé par l'utilisateur est plus petit ou plus grand que le nombre à deviner. En combien d'étapes au plus peut-on deviner le nombre si on connaît le principe de la dichotomie ? Écrire un autre programme qui cherche à deviner le nombre.



Exercice 20.4

On suppose que l'on a un dictionnaire qui contient *n* mots en vrac, sans aucun ordre.

1. Peut-on trouver une méthode plus rapide que la comparaison avec chacun des mots du dictionnaire ?
2. On suppose que l'on cherche au hasard dans ce dictionnaire un mot donné, sans noter les mots déjà essayés. Quelle est la probabilité de trouver le mot du premier coup ? Quelle est la probabilité de trouver le mot au *k*-ème coup, c'est-à-dire d'échouer aux *k* - 1 premiers coups et de réussir au *k*-ème ?
3. Quelle est la probabilité de ne pas encore avoir trouvé le mot après *k* coups ? À partir de combien de coups cette probabilité devient-elle inférieure à 1/2 ?

4. Rechercher sur le Web des informations sur la loi géométrique et sur son espérance et en déduire le nombre moyen de coups nécessaires pour trouver le mot dans le dictionnaire, s'il y est. Le résultat est surprenant : il faut en moyenne *n* coups.
5. Que se passe-t-il si on cherche au hasard le mot, sans noter les mots déjà essayés, et qu'il n'est pas dans la boîte ?

La conversion analogique-numérique

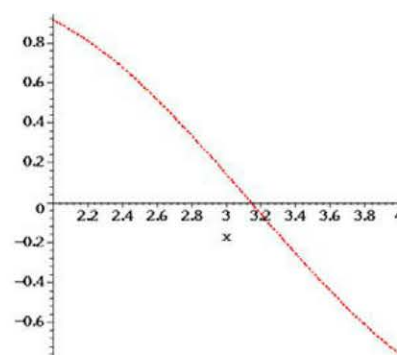
Nous nous intéressons maintenant à un tout autre problème : celui de la conversion analogique-numérique, par exemple d'une tension (voir le chapitre 17).

Il n'est pas difficile de réaliser un circuit qui compare deux tensions et indique laquelle est la plus grande. Il n'est pas non plus trop difficile de réaliser un circuit de conversion numérique-analogique, qui produit une tension donnée par un nombre exprimé en binaire. Beaucoup de convertisseurs analogique-numérique procèdent alors en comparant la tension à numériser successivement à plusieurs valeurs de référence pour, de proche en proche, cerner la valeur de la tension. Ici encore, ces convertisseurs procèdent par dichotomie, en divisant l'espace de recherche par deux à chaque mesure. Cela permet d'atteindre très rapidement une bonne précision, le millième en 10 étapes, le millionième en 20, etc.

Trouver un zéro d'une fonction

Voici encore un autre problème : trouver un zéro d'une fonction f continue et strictement monotone dans un intervalle [a, b], c'est-à-dire résoudre une équation de la forme $f(x) = 0$, $a \leq x \leq b$.

Une méthode consiste à comparer le signe de f (a) et f (b). Si ces deux signes sont identiques, alors la fonction f étant strictement monotone, elle ne s'annule pas sur l'intervalle [a, b]. Sinon, elle s'annule une fois



exactement sur cet intervalle. C'est le cas par exemple de la fonction sinus qui s'annule sur l'intervalle [2,4].

Ici encore, une recherche par dichotomie permet de trouver une valeur approchée de ce zéro. On cherche une valeur approchée à e = 10^{-5} près.

```
e = 1E-5;
a = 2.0;
b = 4.0;
m = (a + b) / 2;
while (b - a > e && Math.abs(Math.sin(m)) > e) {
  if (Math.sin(a) * Math.sin(m) <= 0) {
    b = m;}
  else {
    a = m;}
  m = (a + b) / 2;}
```

Ce programme donne la valeur m = 3,1416015625.

Exercice 20.5

Montrer que ce programme se termine. Montrer que, quand on sort de la boucle, le nombre m est soit une *approximation en x* de la solution, c'est-à-dire un nombre m tel qu'il existe un nombre z tel que $f(z) = 0$ et |m - z| ≤ e, soit une *approximation en y* de la solution, c'est-à-dire un nombre m tel que |f(m)| ≤ e.



Exercice 20.6

L'exercice 20.5 suppose que le calcul de la fonction f sur des nombres à virgule soit exact. Or, on a vu que c'est rarement le cas : les calculs sur les nombres à virgule sont presque toujours des calculs approchés. Donner un exemple dans lequel cet algorithme ne fonctionne pas à cause des approximations sur le calcul de la fonction f. Montrer que si l'erreur sur le calcul de la fonction f est toujours strictement inférieure à e, alors cet algorithme fonctionne toujours.



Exercice 20.7

Si la fonction n'est pas monotone, l'algorithme continue-t-il de se terminer ou risque-t-il de boucler indéfiniment ?



Exercice 20.8

Soit l = b - a la taille de l'intervalle initial. Montrer que, après n itérations, l'intervalle de recherche est de taille l / 2^n. En déduire que le nombre d'itérations nécessaires pour trouver une approximation de la solution est inférieur au logarithme entier de l / e.



Exercice 20.9

Programmer cet algorithme de recherche du zéro d'une fonction de manière récursive.

Ai-je bien compris ?

- Que signifie le mot dichotomie ?
- Combien de comparaisons faut-il faire pour trouver un mot dans un dictionnaire ?
- Quelles sont les autres applications du principe de dichotomie présentées dans ce chapitre ?

Trier

21

Chapitre avancé

Dans une très grande bibliothèque, il faut classer les livres pour les retrouver.

Dans ce chapitre, nous voyons deux algorithmes de tri, ce qui est surtout un prétexte pour nous interroger sur la complexité des algorithmes.

Nous présentons le tri par sélection et le tri par fusion et nous nous interrogeons sur l'efficacité de ces deux algorithmes, en évaluant leur temps d'exécution.



Philippe Flajolet (1948-2011) est un des pionniers de l'analyse de la complexité des algorithmes, c'est-à-dire du temps que dure leur exécution et de la quantité de mémoire qu'elle demande. Il a montré l'utilité, dans ce domaine, de plusieurs théories mathématiques : la combinatoire et le dénombrement, la théorie des probabilités et la théorie des fonctions d'une variable complexe. Il a aussi eu conscience très tôt de l'apport des logiciels de calcul formel pour effectuer les calculs, parfois fastidieux, que cette analyse demande.

Nous avons vu que la recherche d'un élément dans une table est plus rapide quand celle-ci est ordonnée. C'est une chose que nous savons depuis qu'il existe des bibliothèques et des bibliothécaires : même si cela est long et fatigant, il vaut mieux ranger les livres d'une bibliothèque une fois pour toutes, par exemple dans l'ordre alphabétique, plutôt que les laisser en vrac et arpenter des kilomètres de rayonnages à chaque fois que l'on cherche un volume. Cela mène naturellement à un nouveau problème : comment ordonner une table ?

Ce problème est un cas particulier d'un problème plus général : si l'on se donne n objets, par exemple, n nombres, n chaînes de caractères, etc. et une relation $\leq$ réflexive, transitive et totale (voir le chapitre 20), comment trier ces n objets, c'est-à-dire les disposer dans un certain ordre de façon à ce que si un élément a est avant un élément b dans la table alors $a \leq b$?

ALLER PLUS LOIN L'antisymétrie

On ne suppose pas la relation $\leq$ nécessairement antisymétrique, c'est-à-dire que l'on suppose possible d'avoir à la fois $x \leq y$ et $y \leq x$, sans que x et y soient identiques. Cela permet par exemple de trier des points du plan par abscisse croissante. En effet, la relation « avoir une abscisse inférieure ou égale à » est réflexive, transitive et totale, mais elle n'est pas antisymétrique ; par exemple, (1 ; 2) a une abscisse inférieure ou égale à (1 ; 3) et (1 ; 3) a une abscisse inférieure ou égale à (1 ; 2), mais ces deux points sont distincts. Quand on trie par exemple les points (1 ; 2), (2 ; 0), (0 ; 2) et (1 ; 3) par abscisse croissante, deux résultats sont possibles : (0 ; 2), (1 ; 2), (1 ; 3), (2 ; 0) et (0 ; 2), (1 ; 3), (1 ; 2), (2 ; 0).

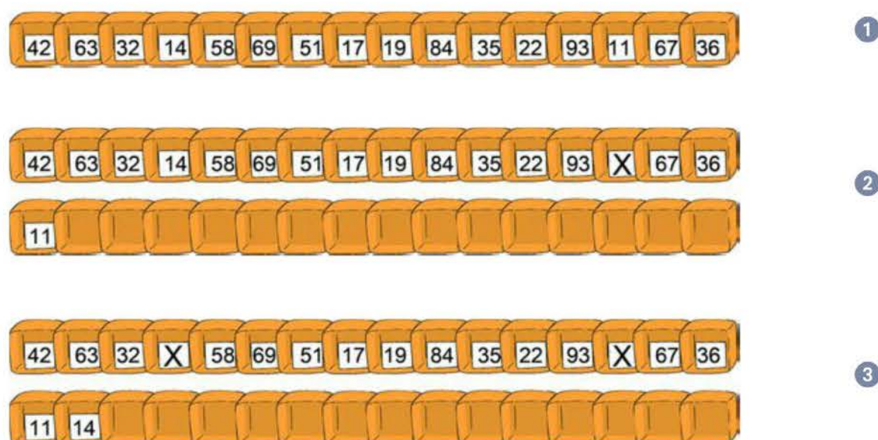
L'importance de ce problème fait que plusieurs dizaines d'algorithmes différents ont été proposés pour trier des objets. Ce chapitre est consacré à deux d'entre eux : le *tri par sélection* et le *tri par fusion*. Comme nous allons le voir, ces deux algorithmes sont d'une efficacité très différente.

Par souci de simplicité, nous supposerons dans tout ce chapitre que les objets à trier sont des nombres entiers et qu'ils sont donnés dans un tableau. Le but d'un algorithme de tri est donc de calculer un nouveau tableau, ou de modifier le tableau initial, de manière à ce qu'il contienne les mêmes nombres que le tableau initial, mais que ces éléments soient ordonnés.

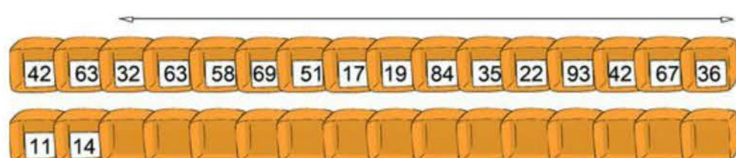
Le tri par sélection

L'algorithme de tri par sélection est volontiers utilisé pour trier des objets, comme des cartes à jouer, des livres, etc. à la main. On commence par chercher, parmi les objets à trier, un élément plus petit que tous les autres. Cet élément sera le premier du tableau trié. On cherche ensuite, parmi ceux qui restent, un élément plus petit que tous les autres, qui sera le deuxième du tableau trié, etc.

Par exemple, pour trier ainsi le tableau ❶, on sélectionne d'abord le plus petit élément, 11, que l'on met au début du tableau résultat et que l'on supprime du tableau à trier (❷). On cherche ensuite le plus petit parmi ceux qui restent, 14, on le met dans le tableau résultat et on le supprime du tableau à trier (❸) et on continue ainsi jusqu'à avoir épuisé le tableau à trier.

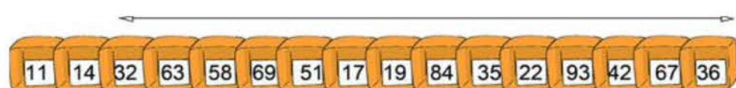


Quand on programme cet algorithme, il faut définir comment exprimer le fait que les cases 3 et 13 du tableau à trier sont désormais vides. Une solution parmi d'autres est d'y mettre les deux derniers éléments, 36 et 67, ou les deux premiers, 42 et 63, de manière à garder les éléments à trier contigus, si bien qu'il suffit de se souvenir des deux extrémités du segment du tableau où se trouvent les éléments à trier.



Pour se souvenir que les éléments à trier sont maintenant dans les cases 2 à 15, et non 0 à 15, du tableau initial, il suffit d'utiliser une variable `i` qui vaut 2 et qui indique à la fois le nombre d'éléments déjà triés et l'indice du tableau où commencent ceux qui restent à trier.

Comme les deux premières cases du tableau initial ne servent plus à rien, on peut les utiliser pour stocker le début du tableau déjà trié, si bien que l'on évite le recours à un tableau auxiliaire.



Si l'on doit trier 16 éléments, rangés dans un tableau dont l'indice varie entre 0 et 15, le tri par sélection se programme de la manière suivante :

```
for (i = 0; i <= 14; i = i + 1) {
  k = i;
```

```
  for (j = i + 1; j <= 15; j = j + 1) {
    if (tab[j] <= tab[k]) {
      k = j;}}
  z = tab[i];
  tab[i] = tab[k];
  tab[k] = z;}
```

L'indice `i` de la boucle principale varie entre 0 et 14. Pour chaque valeur de `i`, on cherche, dans la partie du tableau comprise entre `i` et 15, l'indice `k` d'un élément minimal :

```
k = i;
for (j = i + 1; j <= 15; j = j + 1) {
  if (tab[j] <= tab[k]) {
    k = j;}}
```

puis on échange dans le tableau l'élément d'indice `i` avec celui d'indice `k` :

```
z = tab[i];
tab[i] = tab[k];
tab[k] = z;
```

Quand la boucle est achevée, quinze éléments ont été sélectionnés dans l'ordre croissant et placés dans les quinze premières cases du tableau. Le seizième élément qui reste est plus grand que tous les autres. Le tableau est donc trié.

Exercice 21.1

Effectuer à la main un tri par sélection des tableaux :

57	8	14	26	41	98	3

7	6	5	4	3	2	1

97	83	55	81	20	73	0

2	3	4	5	6	7	1



Exercice 21.2

Dans un tableau `tab` déjà trié, on souhaite insérer un nouvel élément `e` de sorte que le nouveau tableau soit également trié.

1. Proposer un algorithme qui détermine la position à laquelle il faut insérer ce nouvel élément.
2. Si l'on souhaite conserver les éléments triés dans le même tableau `tab`, que faudra-t-il faire avant de pouvoir insérer `e` à sa place ? Dans quel cas cette opération demandera-t-elle beaucoup de temps ?



Exercice 21.3

Si l'on interrompt l'exécution de l'algorithme du tri par sélection après k étapes, on obtient un tableau qui contient les k premiers éléments du tableau final calculé par l'algorithme. Cet algorithme procède en calculant successivement le premier élément du tableau final, puis les deux premiers éléments du tableau final, puis les trois premiers, etc. Un autre algorithme, le *tri par insertion*, trie d'abord le premier élément du tableau initial, puis les deux premiers, puis les trois premiers, etc. Si l'on interrompt l'exécution de l'algorithme du tri par insertion après k étapes, on obtient un tableau qui contient, non les k premiers éléments du tableau final, mais un tableau ordonné qui contient les k premiers éléments du tableau initial.

Chercher sur le Web une description précise de cet algorithme et le programmer.



Exercice 21.4

L'algorithme du *tri à bulles* consiste à trier un tableau en ne s'autorisant qu'à échanger deux éléments consécutifs de ce tableau. On peut démontrer que l'algorithme suivant :

1 chercher deux éléments consécutifs rangés dans le désordre,
2 si deux tels éléments existent, les échanger et recommencer,
3 sinon arrêter,

trie n'importe quel tableau.

1. Effectuer à la main un tri à bulles du tableau :



2. Quel est le temps d'exécution du tri à bulles sur un tableau ordonné ? Était-ce le cas pour le tri par sélection ?
3. Les tableaux sur lesquels le tri à bulles est le moins efficace sont ceux qui sont rangés dans l'ordre décroissant, par exemple :



   Si l'on commence par essayer de placer le nombre n à la position correcte, combien de permutations sont nécessaires pour y arriver ?

   Ensuite, combien de permutations sont nécessaires pour placer $n - 1$ au bon endroit ? Et le nombre $n - 2$?

   Émettre une conjecture sur le nombre total de permutations nécessaires pour trier un tableau de taille n rangé initialement en ordre décroissant.

   Démontrer cette conjecture par récurrence.
4. Proposer une description plus précise de cet algorithme : l'étape « chercher deux éléments consécutifs rangés dans le désordre » pouvant être traitée de façons assez variées. Programmer cet algorithme.

Le tri par fusion

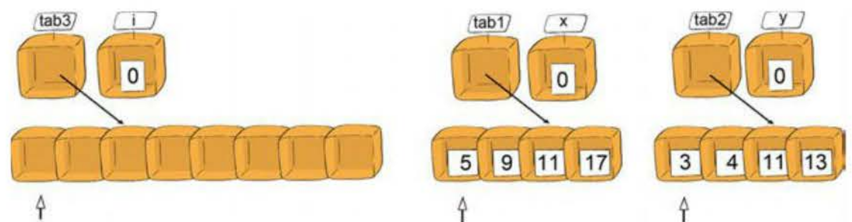
Comme nous le verrons plus loin, les algorithmes de tri par sélection, par insertion ou à bulles sont très lents. L'algorithme de *tri par fusion* fait la même chose que ces trois algorithmes, mais beaucoup plus rapidement.

Il est plus simple de présenter cet algorithme avec un tableau dont la taille est une puissance de 2, comme 8, 16, 32, etc. Cela n'est pas réellement une limitation, car si l'on veut trier 25 éléments par exemple, il suffit d'ajouter 7 éléments avec une très grande valeur dans le tableau à trier, puis de supprimer les 7 derniers éléments du tableau trié.

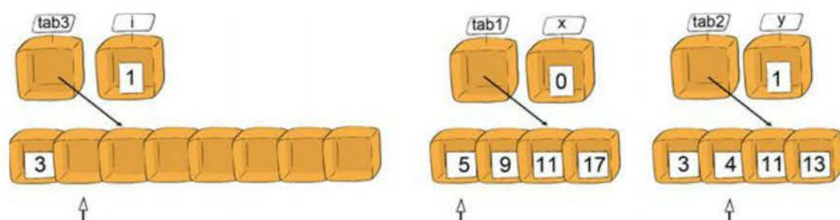
Le cœur de l'algorithme est un autre algorithme qui permet de fusionner deux tableaux triés. Par exemple, si le tableau `tab1` contient les éléments 5, 9, 11 et 17, dans cet ordre, et si le tableau `tab2` contient les éléments 3, 4, 11 et 13, alors leur fusion est un tableau qui contient ces huit éléments dans l'ordre, c'est-à-dire 3, 4, 5, 9, 11, 11, 13 et 17. Pour fusionner deux tels tableaux, le principe général est de transférer les différents éléments dans un troisième tableau dans l'ordre croissant, l'élément suivant étant toujours forcément situé au début d'un des deux tableaux initiaux. On peut pour cela utiliser le programme suivant :

```
x = 0;
y = 0;
for (i = 0; i <= 7; i = i + 1) {
  if ((x <= 3 && y <= 3 && tab1[x] <= tab2[y]) || (y == 4)) {
    tab3[i] = tab1[x];
    x = x + 1;}
  else {
    tab3[i] = tab2[y];
    y = y + 1;}}
```

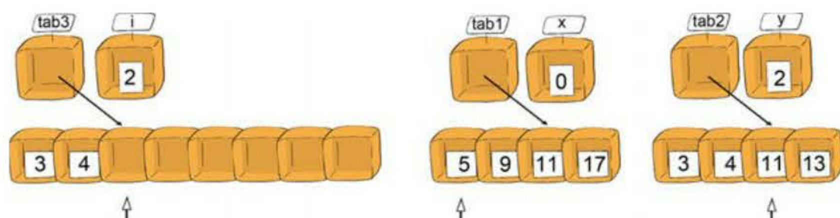
Ce programme est essentiellement constitué d'une boucle qui détermine l'un après l'autre les éléments du tableau résultat `tab3`. L'indice `i` désigne la prochaine case à remplir dans ce tableau. Au départ, `tab3` est vide et l'indice `i` vaut 0. Les indices `x` et `y` désignent le prochain élément des tableaux `tab1` et `tab2` non encore recopiés.



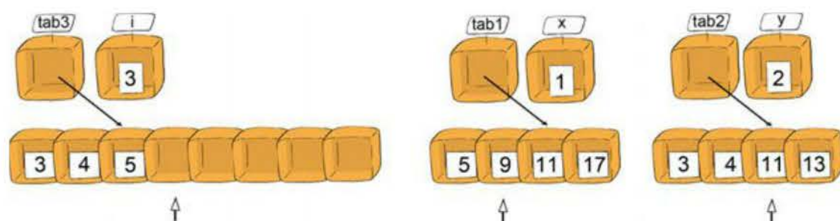
Au cours du premier tour de boucle, on compare l'élément `x` du tableau `tab1` à l'élément `y` du tableau `tab2`. C'est ce second élément qui est plus petit. On le recopie dans la case `i` du tableau `tab3` et on augmente de 1 la valeur des indices `y` et `i`.



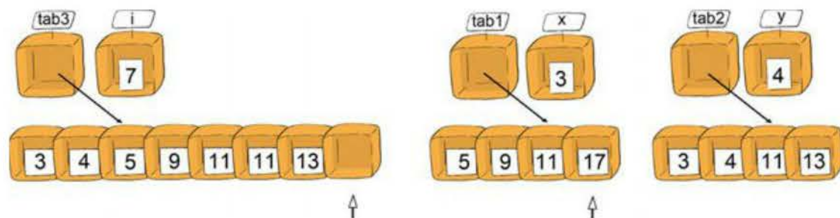
C'est encore l'élément `y` du tableau `tab2` qui est le plus petit ; c'est encore lui qu'on recopie dans le tableau `tab3` au tour suivant de la boucle.



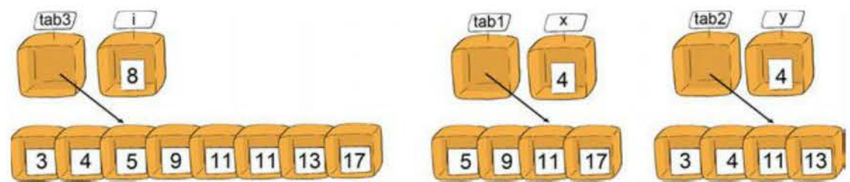
C'est désormais l'élément `x` du tableau `tab1` qui est le plus petit ; c'est lui qu'on recopie dans le tableau `tab3` au tour suivant de la boucle.



Et on continue ainsi jusqu'au septième tour de la boucle, où l'on recopie le dernier élément du tableau `tab2` dans le tableau `tab3`. Il n'est alors plus nécessaire de comparer les éléments plus petits et non encore recopiés des deux tableaux. C'est dans le premier qu'on prendra désormais tous les éléments à recopier.



Au huitième tour de boucle, on recopie le dernier élément du tableau `tab1`.



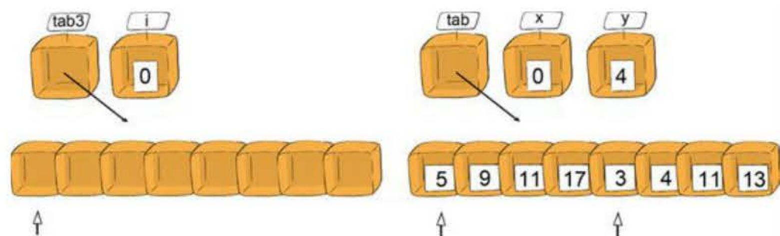
et la construction du tableau `tab3` est achevée.



Exercice 21.5

Modifier ce programme pour fusionner trois tableaux, chacun étant de taille 4.

Au lieu d'utiliser deux tableaux `tab1` et `tab2`, on peut aussi utiliser deux segments du même tableau `tab` par exemple, le segment qui va de la case 0 à la case 3 et le segment qui va de la case 4 à la case 7.



Le programme s'écrit alors de la manière suivante :

```
x = 0;
y = 4;
for (i = 0; i <= 7; i = i + 1) {
  if ((x <= 3 && y <= 7 && tab[x] <= tab[y]) || (y == 8)) {
    tab3[i] = tab[x];
    x = x + 1;}
  else {
    tab3[i] = tab[y];
    y = y + 1;}}
```

Maintenant, comment utiliser cet algorithme de fusion pour trier un tableau ? On commence avec un tableau non trié, dont le nombre d'éléments est une puissance de deux, par exemple 16. Chaque case du tableau est un mini-segment formé d'une case unique. Puisqu'il ne comporte qu'une case, chacun de ces mini-segments est ordonné. On les regroupe alors deux par deux et on fusionne, l'une après l'autre, ces huit paires de segments.

42	63	32	14	58	69	51	17	19	84	35	22	93	11	67	36
42	63	14	32	58	69	17	51	19	84	22	35	11	93	36	67

On obtient alors huit segments ordonnés de deux cases. On les regroupe deux par deux et on fusionne l'une après l'autre ces quatre paires de segments de deux éléments.

42	63	14	32	58	69	17	51	19	84	22	35	11	93	36	67
14	32	42	63	17	51	58	69	19	22	35	84	11	36	67	93

On obtient alors quatre segments ordonnés de quatre cases. On les regroupe deux par deux et on fusionne l'une après l'autre ces deux paires de segments de quatre éléments.

14	32	42	63	17	51	58	69	19	22	35	84	11	36	67	93
14	17	32	42	51	58	63	69	11	19	22	35	36	67	84	93

On obtient alors deux segments ordonnés de huit cases et on fusionne cette paire de segments de huit éléments.

14	17	32	42	51	58	63	69	11	19	22	35	36	67	84	93
14	17	32	42	51	58	63	69	11	19	22	35	36	67	84	93

et on obtient le tableau ordonné.

Exercice 21.6

Effectuer à la main un tri par fusion des tableaux :

66	11	20	43	80	22	17	61

28	10	31	38	12	66	44	62

8	7	6	5	4	3	2	1

5	1	7	3	6	2	8	4

Le programme s'écrit ainsi :

```
s = 1;
while (s <= 15) {
  b = 0;
  x = 0;
  y = s;
  for (i = 0; i <= 15; i = i + 1) {
    if ((x < b + s && y < b + 2 * s && tab[x] < tab[y])
        || (y == b + 2 * s)) {
      tab1[i] = tab[x];
      x = x + 1;}
```

```
    else {
      tab1[i] = tab[y];
      y = y + 1;}
    if (x == b + s && y == b + 2 * s) {
      b = b + 2 * s;
      x = b;
      y = b + s;}}
  for (i = 0; i <= 15; i = i + 1) {
    tab[i] = tab1[i];}
  s = s * 2;}
```

Dans la boucle qui va des lignes 6 à 16, on parcourt le tableau `tab` et on fusionne des petits segments de taille `s` en segments de taille 2 `s`. On commence par poser `b` = 0 et par fusionner les segments [`b`, `b + s` - 1] et [`b + s`, `b` + 2 `s` - 1], c'est-à-dire [0, `s` - 1] et [`s`, 2 `s` - 1], en choisissant, comme précédemment, les éléments alternativement dans un segment et dans l'autre, en augmentant d'une unité la variable `x` ou bien la variable `y`, selon le segment choisi, jusqu'à ce que `x` soit égal à `b + s` et `y` à `b` + 2 `s`. À ce moment, on a fini de fusionner les deux premiers segments, on pose `b` = 2 `s` et on continue, dans la même boucle, à fusionner les deux segments suivants : [`b`, `b + s` - 1] et [`b + s`, `b` + 2 `s` - 1], c'est-à-dire [2 `s`, 3 `s` - 1] et [3 `s`, 4 `s` - 1], etc.

Quand cette boucle est achevée, le tableau `tab1` est formé de segments triés de taille 2 `s`. On le recopie dans le tableau `tab`, on multiplie `s` par 2 et on recommence à fusionner les segments de taille supérieure : c'est le rôle de la boucle `while` la plus externe.

Exercice 21.7

Modifier le programme donné pour pouvoir effectuer un tri par fusion

1. d'un tableau dont la taille est une puissance de 2 quelconque,
2. d'un tableau de taille quelconque en utilisant la méthode proposée au début de cette section pour se ramener à une taille qui est une puissance de 2.

L'efficacité des algorithmes

Comme on l'a déjà vu plusieurs fois, pour traiter un même problème, il existe souvent plusieurs algorithmes. Par exemple, on peut trier un tableau par sélection, par fusion, etc. Quand on doit choisir parmi plusieurs algorithmes, l'un des critères est celui du temps d'exécution. Pour un logiciel interactif, par exemple, un temps de réponse court est un élément essentiel du confort de l'utilisateur. De même, certains programmes industriels doivent être utilisés un grand nombre de fois dans un délai très court et même

un programme qui n'est exécuté qu'une seule fois, par exemple un programme de simulation écrit pour tester une hypothèse de recherche, est inutilisable s'il demande des mois ou des années de calcul.

En général, on cherche à déterminer comment le temps d'exécution d'un algorithme varie en fonction de la taille des données. Le temps d'exécution d'une recherche en table dépend de la taille de cette table. Comme on l'a vu, selon l'algorithme, ce temps peut être proportionnel à la taille de la table ou au logarithme de cette taille. De même, quand on s'interroge sur l'efficacité d'un algorithme de tri, on cherche à comprendre comment le temps d'exécution de cet algorithme varie en fonction du nombre d'objets à trier.

L'évaluation du temps mis par un algorithme pour s'exécuter est un domaine de recherche à part entière, car elle se révèle quelquefois très difficile. Néanmoins, dans de nombreux cas, cette évaluation peut se faire en appliquant quelques règles simples.

SAVOIR-FAIRE S'interroger sur l'efficacité d'un algorithme

- Une affectation ou l'évaluation d'une expression ont un temps d'exécution petit. Cette durée constitue souvent l'unité de base dans laquelle on mesure le temps d'exécution d'un algorithme.
- Le temps pris pour effectuer une séquence `p q` est la somme des temps pris pour exécuter les instructions `p` puis `q`.
- Le temps pris pour exécuter un test `if (b) p else q` est inférieur ou égal au maximum des temps pris pour exécuter les instructions `p` et `q`, plus une unité qui correspond au temps d'évaluation de l'expression `b`.
- Le temps pris pour exécuter une boucle `for (i = 1; i <= m; i = i + 1) p` est `m` fois le temps pris pour exécuter l'instruction `p` si ce temps ne dépend pas de la valeur de i. En particulier, quand deux boucles sont imbriquées, le corps de la boucle interne est répété à cause de cette boucle, mais aussi parce qu'elle-même est répétée dans son intégralité. Ainsi, si les deux boucles sont répétées respectivement m et m' fois, alors le corps de la boucle interne est exécuté $m \times m'$ fois en tout. Quand le temps d'exécution du corps de la boucle dépend de la valeur de l'indice i, le temps total d'exécution de la boucle est la somme des temps d'exécution du corps de la boucle pour chaque valeur de i. Quand le nombre de répétitions d'une boucle ne dépend pas des entrées de l'algorithme, le temps pris pour exécuter cette boucle est une constante.
- Le cas des boucles `while` est plus complexe à traiter puisque le nombre de répétitions n'est en général pas connu a priori.

Exercice 21.8 (avec corrigé)

Comment le temps d'exécution des algorithmes exprimés par les programmes suivants varie-t-il en fonction de n ?

```
n = Isn.readInt();
for (i = 0; i <= 10; i = i + 1) {
  System.out.println(i * n);}
```

Cet algorithme affiche la table de multiplication de n. *La boucle est toujours exécutée 10 fois, quel que soit* n, *et donc le temps d'exécution ne dépend pas de l'entrée* n.

```
n = Isn.readInt();
for (i = 1; i <= n; i = i + 1) {
  System.out.println(i * i);}
```

Cet algorithme affiche la suite des carrés des nombres entiers jusqu'à n^2. *La boucle est exécutée* n *fois et le temps d'exécution est donc proportionnel à* n.

```
n = Isn.readInt();
for (i = 1; i <= n; i = i + 1) {
  for (j = 1; j <= n; j = j + 1) {
    System.out.println(i * j);}}
```

Cet algorithme construit une table de multiplication pour tous les entiers de 1 à n *en donnant tous leurs multiples jusqu'au* n*-ième. Il comprend deux boucles imbriquées, chacune effectuant* n *répétitions de son corps. Le temps d'exécution total est donc proportionnel à* n^2.

Exercice 21.9

Comment le temps d'exécution de l'algorithme exprimé par le programme suivant varie-t-il en fonction de *n* ?

```
n = Isn.readInt();
for (i = 1; i <= n; i = i + 1) {
  System.out.println(i * 2);}
for (j = 1; j <= n; j = j + 1) {
  System.out.println(3 * j);}
```

L'efficacité des algorithmes de tri par sélection et par fusion

Pour évaluer le temps que demande l'algorithme de tri par sélection pour trier un tableau, on compte le nombre de fois qu'est exécuté le corps de la boucle la plus interne :

```
if (tab[j] <= tab[k]) {
  k = j;}
```

Cette instruction est exécutée 15 fois la première fois que la boucle interne est exécutée, puis 14 fois, puis 13 fois, …, puis 1 fois. Au total, cette instruction est exécutée 15 + 14 + … + 1 = 120 fois.

Plus généralement, quand on trie un tableau de n éléments, cette instruction est exécutée $(n - 1) + (n - 2) + \ldots + 1$ fois, c'est-à-dire $n\,(n - 1) / 2 = (1 / 2)\,n^2 - (1 / 2)\,n$ fois. Dans cette évaluation, le terme $(1 / 2)\,n$ devient beaucoup plus petit que $(1 / 2)\,n^2$ quand n devient grand et on peut donc le négliger. De même, on peut négliger le temps pris pour échanger les éléments d'indices `i` et `k` du tableau, car cette opération est répétée n fois et elle prend donc un temps proportionnel à n, ce qui peut être négligé devant $(1 / 2)\,n^2$. Au bout du compte, l'information qu'il est important de retenir est que le temps d'exécution de cet algorithme est quasiment proportionnel au carré du nombre d'éléments du tableau à trier.

L'algorithme de tri par fusion est lui aussi composé de deux boucles imbriquées. On pourrait donc s'attendre à ce que le temps d'exécution de cet algorithme soit proportionnel au carré du nombre d'éléments à trier. Et de fait, la boucle `for` la plus interne, qui fusionne tous les segments de taille `s` en des segments de taille 2 `s`, demande un temps proportionnel au nombre n d'éléments à trier puisqu'elle est exécutée n fois, son indice variant de 0 à 15 dans notre exemple et de 0 à $n - 1$ dans le cas général. Cependant, la boucle `while` la plus externe est exécutée un nombre de fois qui est proportionnel, non au nombre n d'éléments à trier, mais au logarithme entier de ce nombre car, à chaque fois, on double la taille `s` des segments fusionnés. Ainsi, dans notre exemple, `s` vaut successivement 1, 2, 4 et 8 et la boucle est exécutée 4 fois, ce qui est le logarithme entier de 16. Ainsi, le temps nécessaire pour trier un tableau de taille n est proportionnel, non pas au carré de n, mais au produit de n par son logarithme entier : $n \times \text{elog}\ n$.

Prenons un exemple : pour trier un tableau d'un million d'éléments, l'algorithme de tri par sélection demande un temps de l'ordre de $n^2 = (10^6)^2 = 10^{12}$, mille milliards. Le logarithme entier de 10^6 est 20 puisque $10^6 / 2^{20} = 0{,}953\ldots$ Le temps demandé par l'algorithme de tri par fusion est donc de l'ordre de $10^6 \times 20$, vingt millions. Le tri par fusion est plus rapide que le tri par sélection d'un facteur de l'ordre de 50 000 dans cet exemple. Si un ordinateur exécute le corps de la boucle interne en une milliseconde, un algorithme mettra un temps de l'ordre de vingt secondes et l'autre d'un million de secondes, soit un peu plus de dix jours, pour trier ce tableau.

ALLER PLUS LOIN Le tri par fusion programmé récursivement

Comme de nombreux autres algorithmes, le tri par fusion peut se programmer de manière récursive. Il peut se décrire simplement en disant que pour trier un segment d'un tableau de plus de deux éléments, il faut trier la première moitié, trier la seconde, puis fusionner les deux segments obtenus. Cela mène à la fonction suivante, qui trie le segment du tableau `tab`, allant de `n` à `p` :

```
static void tri (int [] tab, int n, int p) {
  int k,i;
  int [] tab1;
  if (n < p) {
    k = (n + p)/2;
    tri(tab,n,k);
    tri(tab,k+1,p);
    tab1 = new int [p+1];
    fusion(tab,n,k,k+1,p,tab1,n);
    for (i = n; i <= p; i = i + 1) {
      tab[i] = tab1[i];}}}
```

Outre les deux appels récursifs à la fonction `tri` elle-même, ce programme utilise un appel à la fonction `fusion` qui fusionne deux segments du tableau `tab`, allant de `n` à `m` et de `p` à `q` en mettant le résultat dans le tableau `tab1`, à partir de l'indice `i`. Cette fonction peut elle-même se programmer avec une boucle, ou alors récursivement :

```
static void fusion (int [] tab,int x,int n,int y,int p,int tab1
[],int i) {
  if (x <= n && (y > p || tab[x] < tab[y])) {
    tab1[i] = tab[x];
    fusion(tab,x+1,n,y,p,tab1,i+1);}
  else if (y <= p) {
    tab1[i] = tab[y];
    fusion(tab,x,n,y+1,p,tab1,i+1);}}
```

Dans le programme principal, il ne reste plus qu'à remplir le tableau `tab` avec les nombres à trier et appeler la fonction `tri` :

```
tri(tab,0,15);
```

Ces deux manières de programmer l'algorithme de tri par fusion illustrent, à nouveau, la différence entre algorithme et programme, discutée au chapitre 18. Même en restant dans le même langage de programmation, l'algorithme de tri par fusion peut s'incarner dans des programmes très différents, selon que l'on utilise des boucles ou la récursivité pour l'exprimer.

Ai-je bien compris ?

- Quels sont les principaux algorithmes de tri présentés dans ce chapitre ?
- Qu'est-ce que la complexité d'un algorithme ?
- Quel est l'algorithme de tri le plus rapide parmi ceux présentés dans ce chapitre ?

Parcourir un graphe

22

CHAPITRE AVANCÉ



Joseph Sifakis (1946-) est un chercheur français d'origine grecque qui a reçu le prix Turing en 2007, avec Edmund Clarke et Allen Emerson, pour la *méthode d'énumération et de vérification des modèles*. Cette méthode se fonde sur une description des systèmes informatiques par des systèmes à états et transitions et sur une analyse des états accessibles dans ces systèmes, qui s'inspire des algorithmes de parcours de graphes. Il est le premier scientifique français à avoir reçu ce prix.

Où on trouve enfin la sortie du labyrinthe.

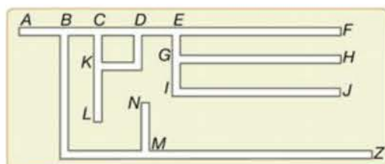
Dans ce dernier chapitre, nous voyons un algorithme de parcours de graphe qui permet par exemple, de chercher la sortie d'un labyrinthe, en évitant de tourner en rond en conservant à chaque étape une liste des chemins à prolonger.

Diverses variantes de cette méthode permettent de parcourir le graphe en profondeur ou en largeur.

Partant de l'exemple des labyrinthes, nous définissons la notion de graphe. Cette notion est importante car les graphes permettent de modéliser nombre de problèmes, par exemple de nombreux jeux, où les sommets représentent les états possibles du jeu et les arêtes représentent les coups possibles.

Un algorithme de parcours de graphe est, à peu de choses près, un algorithme qui permet de trouver la sortie d'un labyrinthe. Si la systématisation et l'étude de ces algorithmes est récente, la notion de graphe elle-même est très ancienne, puisqu'on a retrouvé des labyrinthes dessinés dans des tombes datant de la préhistoire.

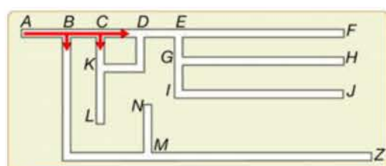
La liste des chemins à prolonger



Commençons par un exemple.

On entre dans ce labyrinthe par le point *A* en haut à gauche, on avance de carrefour en carrefour, et on cherche la sortie *Z*, en bas à droite.

En partant de l'entrée *A*, on n'a guère d'autre choix que d'avancer droit devant soi jusqu'au carrefour *B*, où deux possibilités se présentent : tourner à droite ou continuer tout droit. Si on continue tout droit, vers le carrefour *C*, et que cela se révèle un mauvais choix, il faudra plus tard explorer l'autre possibilité : revenir en *B* et prendre la galerie qui mène au carrefour *M*. Tout le temps que dure l'exploration de la première possibilité, on doit donc garder en mémoire que la seconde reste à explorer. Au carrefour *C* se pose à nouveau le choix entre deux galeries qui mènent l'une au carrefour *D* et l'autre au carrefour *K*. Si on prend la galerie qui mène au carrefour *D*, on doit garder en mémoire que, en plus du chemin *A-B-C-D*, il y a désormais deux autres chemins dont on doit explorer les prolongements : *A-B-M* et *A-B-C-K*.



À chaque étape de son exploration, il faut donc se souvenir d'une *liste de chemins à prolonger*. On peut se représenter cette liste comme les cailloux du Petit Poucet ou le fil d'Ariane qui, en marquant le chemin parcouru, mettent en évidence ceux qui restent à explorer lorsqu'il faudra revenir sur ses pas. Au départ, la liste contient un chemin unique, formé d'un carrefour unique, qui est l'entrée du labyrinthe :

```
A
```

Cette liste des chemins à prolonger devient ensuite :

```
A-B
```

puis

```
A-B-C
A-B-M
```

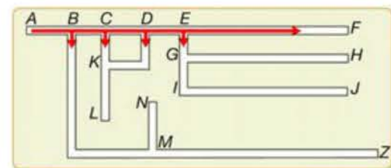
On choisit alors un chemin ou l'autre et on le prolonge. Si on choisit le premier, cette liste devient :

```
A-B-C-D
A-B-C-K
A-B-M
```

Si on choisit de prolonger encore le premier chemin, elle devient :

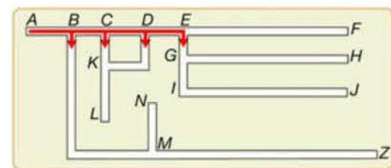
```
A-B-C-D-E
A-B-C-D-K
A-B-C-K
A-B-M
```

puis :




```
A-B-C-D-E-F
A-B-C-D-E-G
A-B-C-D-K
A-B-C-K
A-B-M
```

Si on choisit encore de prolonger le premier chemin *A-B-C-D-E-F*, on rencontre une situation nouvelle, car le carrefour *F* est une impasse. On supprime donc simplement le chemin *A-B-C-D-E-F* de la liste des chemins à prolonger, qui devient donc :



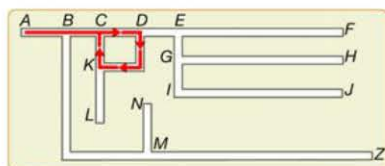

```
A-B-C-D-E-G
A-B-C-D-K
A-B-C-K
A-B-M
```

et on entame alors l'exploration d'un autre chemin de cette liste, par exemple *A-B-C-D-E-G*.

De manière générale, à chaque étape, on transforme la liste de chemins à prolonger en choisissant un chemin *c*, par exemple le premier de la liste, et en le remplaçant par tous les chemins obtenus en ajoutant à *c* un carrefour accessible depuis son dernier carrefour. Un cas particulier est celui où ce dernier carrefour est une impasse. Dans ce cas, on remplace le chemin *c* par zéro chemin, c'est-à-dire qu'on le supprime de la liste des chemins à prolonger.

Il y a cependant deux exceptions. Si le dernier carrefour du chemin *c* est la sortie du labyrinthe, l'algorithme s'arrête et retourne le chemin *c* qui va de l'entrée à la sortie du labyrinthe. Si la liste des chemins à prolonger est vide, l'algorithme s'arrête également : tous les chemins ont été explorés sans succès : il n'y en a aucun qui conduise de l'entrée à la sortie.

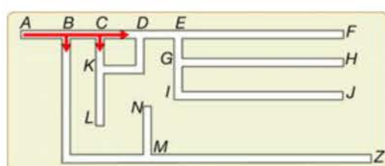
Éviter de tourner en rond



Cette méthode fonctionne pour certains labyrinthes, mais pas pour tous. Dans le cas de notre exemple précédent, par exemple, elle peut échouer, ou plus précisément se lancer dans des calculs infinis, sans jamais trouver le chemin *A-B-M-Z* qui mène de l'entrée à la sortie. Au carrefour *C*, en effet, on peut continuer vers le carrefour *D*, puis prendre à droite vers le carrefour *K* et encore à droite, ce qui ramène au carrefour *C*. On peut alors à nouveau prendre à droite vers le carrefour *D*, et ainsi de suite à l'infini : ce labyrinthe comporte un cycle. En appliquant la méthode décrite, on peut remplacer le chemin *A-B-C-D*, par des chemins parmi lesquels figure *A-B-C-D-K*, que l'on remplace à son tour par des chemins parmi lesquels figure *A-B-C-D-K-C*, que l'on remplace à son tour par des chemins parmi lesquels figure *A-B-C-D-K-C-D*, et ainsi de suite à l'infini.

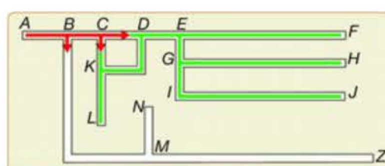
Pour trouver la sortie d'un labyrinthe, il faut donc certes explorer systématiquement tous les chemins possibles, mais aussi éviter de tourner en rond.

En fait, il n'est même pas nécessaire que le labyrinthe comporte un cycle pour que la méthode décrite se lance dans un calcul infini. On a dit que, arrivé au carrefour *C*, on a deux possibilités : continuer tout droit vers le carrefour *D* ou tourner à droite vers le carrefour *K*. *Stricto sensu*, il y a une troisième possibilité : faire demi-tour et aller vers *B*. Si on inclut cette possibilité, la méthode peut se lancer dans un calcul infini, en allant de *B* à *C*, puis de *C* à *B*, etc.



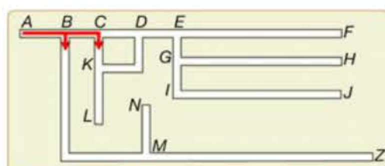
Il y a plusieurs manières d'éviter que cette méthode tourne ainsi en rond. La plus simple consiste à ne pas ajouter à la liste les chemins obtenus en ajoutant un carrefour déjà inclus dans le chemin *c* et qui forment donc un cycle.

Cette méthode est correcte, mais il est possible de faire mieux. En effet, comme on l'a vu, quand on arrive au carrefour *D*, la liste des chemins à prolonger est :




```
A-B-C-D
A-B-C-K
A-B-M
```

L'exploration du chemin *A-B-C-D* est longue puisqu'il faut parcourir toute la partie du labyrinthe en vert sur la figure.



Cependant, cette longue exploration ne mène qu'à des impasses, si bien que la liste des chemins à prolonger devient finalement :

```
A-B-C-K
A-B-M
```

À partir de *K*, on ne peut pas aller en *C*, car ce carrefour appartient au chemin *A-B-C-K*, mais rien n'empêche d'aller en *D* ou en *L*, si bien qu'on doit explorer à nouveau toute la partie du labyrinthe en vert, alors qu'en arrivant en *K*, on pourrait se rendre compte que l'on est déjà passé par là et qu'il n'est pas nécessaire de continuer.

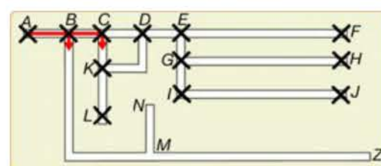
Pour éviter à la fois de tourner en rond et d'explorer plusieurs fois les mêmes parties du labyrinthe, une solution consiste à marquer les carrefours que l'on visite pour ne jamais repasser par eux. L'algorithme auquel on aboutit ainsi utilise donc, d'une part, une liste des chemins à prolonger et, d'autre part, une *liste des carrefours déjà visités*.

Ainsi, au départ, la liste des chemins à prolonger contient un seul chemin formé d'un seul carrefour, qui est le point d'entrée du labyrinthe :

```
A
```

et la liste des carrefours déjà visités est vide. Puis, à chaque étape, on transforme la liste de chemins à explorer de la manière suivante : on choisit un chemin *c* dans la liste des chemins à prolonger. Si le dernier carrefour *x* de ce chemin appartient à la liste des carrefours déjà visités, on supprime simplement le chemin *c*. Sinon, on le remplace par les chemins obtenus en ajoutant à *c* un carrefour accessible depuis *x* et on ajoute *x* à la liste des carrefours déjà visités.

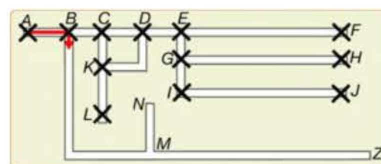
Ainsi, dans notre exemple, quand on a fini l'exploration du chemin *A-B-C-D*, la liste des chemins à prolonger devient :




```
A-B-C-K
A-B-M
```

et la liste des carrefours déjà visités contient *A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I*, *J*, *K* et *L*.

K est dans la liste des carrefours déjà visités ; il est donc inutile d'y retourner et la liste des chemins à prolonger devient :




```
A-B-M
```

Il est facile de montrer que cet algorithme se termine, car le nombre de carrefours déjà visités augmente à chaque étape et il ne peut pas augmenter au-delà du nombre total de carrefours du labyrinthe.

Il est, en revanche, plus délicat de montrer que cet algorithme trouve toujours un chemin vers la sortie quand un tel chemin existe, car il faut montrer qu'en évitant ainsi l'exploration de nombreux chemins, on n'oublie pas d'explorer celui qui mène de l'entrée à la sortie. On montre, pour cela, que s'il existe au départ, dans la liste des chemins à prolonger, un chemin *prolongeable vers la sortie*, alors cette propriété est préservée à chaque étape de l'exé-

cution de l'algorithme : c'est un invariant. Un chemin est dit *prolongeable vers la sortie* s'il se termine par un carrefour *y* et s'il existe un chemin formé de carrefours distincts et non encore visités, qui mène de *y* à la sortie.

L'étape élémentaire de l'algorithme consiste à transformer la liste des chemins à prolonger en remplaçant un chemin *c* par tous les chemins obtenus en lui ajoutant un carrefour accessible depuis son dernier carrefour *x*. On ajoute alors *x* à la liste des carrefours visités. Si le chemin *c* est lui-même prolongeable vers la sortie, alors il y a évidemment, parmi les nouveaux chemins, un qui est prolongeable vers la sortie. Si, en revanche, *c* n'est pas prolongeable vers la sortie, alors un autre chemin *c'* de la liste l'est. Il faut montrer que ce chemin reste prolongeable vers la sortie, bien qu'à cette étape on ajoute *x* à la liste des carrefours déjà visités. Cela est dû au fait que, comme il n'y a pas de chemin formé de carrefours distincts et non encore visités qui mène du carrefour *x* à la sortie, le carrefour *x* ne peut appartenir à aucun chemin formé de carrefours distincts et non encore visités qui mène du dernier carrefour de *c'* à la sortie. Le chemin *c'* reste donc prolongeable vers la sortie bien que l'on ajoute *x* à la liste des carrefours visités.

La recherche en profondeur et la recherche en largeur

La méthode décrite au paragraphe précédent est encore incomplète, car elle n'indique pas dans quel ordre on doit traiter les chemins à prolonger. Or, selon l'ordre que l'on choisit, on aboutit à différents algorithmes. La manière la plus simple de procéder est celle que nous avons employée dans les exemples : on choisit toujours de traiter le premier chemin de la liste et, quand on le remplace par les chemins obtenus en lui ajoutant un carrefour accessible depuis son dernier carrefour, on met ces chemins au début de la liste également. Par exemple, quand on avait la liste de chemins à prolonger :

```
A-B-C
A-B-M
```

on a choisi le premier de la liste, *A-B-C*, et on l'a prolongé en deux chemins *A-B-C-D* et *A-B-C-K* que l'on a placés au début de la liste :

```
A-B-C-D
A-B-C-K
A-B-M
```

puis on a choisi à nouveau le premier de la liste : *A-B-C-D*.

Cet algorithme s'appelle l'algorithme de *parcours en profondeur* ou *dfs* (*depth-first search*), car on explore d'abord en profondeur les prolongements possibles du chemin *A-B-C*, avant de commencer à explorer ceux du second, *A-B-M*, si jamais la première exploration échoue. Cette manière de faire est la plus naturelle : c'est à la fois la plus simple à programmer et celle que l'on utilise spontanément quand on est perdu dans un labyrinthe.

Une autre manière de faire est de traiter les chemins plus équitablement : pour cela, on choisit toujours le chemin le plus court. Ainsi, quand on a remplacé le chemin *A-B-C* par les deux chemins *A-B-C-D* et *A-B-C-K*, on choisit, non l'un de ces deux-là, mais le chemin *A-B-M* qui est plus court. Une manière simple de faire cela est de mettre les deux chemins *A-B-C-D* et *A-B-C-K* à la fin, et non au début, de la liste des chemins à prolonger :

```
A-B-M
A-B-C-D
A-B-C-K
```

puis d'explorer le premier chemin de la liste : *A-B-M* et de mettre les deux chemins obtenus à la fin de la liste des chemins à prolonger :

```
A-B-C-D
A-B-C-K
A-B-M-N
A-B-M-Z
```

Cet algorithme, qui s'appelle l'algorithme de *parcours en largeur* ou *bfs* (*breadth-first search*), a l'avantage de trouver le chemin le plus court qui va de l'entrée à la sortie du labyrinthe, ou l'un parmi les plus courts s'il y en a plusieurs.

ALLER PLUS LOIN Le chemin de poids minimal

On peut aussi prendre en compte la longueur des galeries. On obtient alors un troisième algorithme qui cherche un chemin vers la sortie *de poids minimal*, le *poids* étant une grandeur abstraite associée à chaque galerie, par exemple sa longueur. Il suffit pour cela de toujours traiter en premier, dans la liste des chemins à prolonger, celui dont le poids est le plus petit.

Le parcours d'un graphe

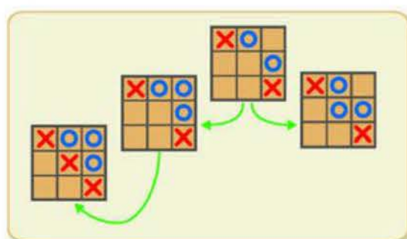
Ces algorithmes sont bien entendu utilisés pour résoudre de nombreux problèmes autres que la recherche de la sortie d'un labyrinthe. De manière générale, ils servent pour parcourir des *graphes*. Un graphe est simplement un ensemble d'objets, que l'on appelle des *sommets*, et un ensemble d'*arêtes*, chaque arête reliant deux sommets entre eux. Dans le cas d'un labyrinthe, les sommets sont les carrefours et les arêtes les galeries qui relient les carrefours.

D'autres exemples de graphes sont les cartes routières ou ferroviaires : les programmes qui trouvent un chemin allant d'une ville à une autre, ou

d'une gare à une autre, utilisent des algorithmes de parcours de graphe, cherchant en général des chemins de poids minimaux.

Les labyrinthes que l'on a parcourus sont des graphes *non orientés*, c'est-à-dire que la même arête relie le sommet A au sommet B et le sommet B au sommet A. Dans d'autres graphes, dits *orientés*, une arête relie soit A à B, soit B à A, et si on veut relier à la fois A à B et B à A, il faut utiliser deux arêtes. On aurait, par exemple, dû utiliser un graphe orienté, s'il y avait eu des sens uniques dans les labyrinthes dont on cherchait la sortie. Les algorithmes étudiés précédemment s'étendent sans difficulté aux graphes orientés.

États et transitions



Beaucoup d'objets peuvent se *modéliser* comme des graphes, en particulier, toutes les situations qui peuvent se décrire à l'aide d'un ensemble d'états et d'un ensemble de transitions entre ces états. C'est le cas de nombreux jeux, comme le Solitaire ou le Tic-tac-toe. La figure ci-contre représente un tout petit bout du graphe où le joueur bleu a deux choix pour placer un rond, l'un des deux le conduisant à perdre la partie.

Un *état du plateau* indique quelle pièce se trouve sur quelle case et les règles du jeu définissent des *transitions* possibles d'un état à un autre. On peut alors définir un graphe dont les sommets sont les états du plateau et les arêtes les coups possibles. Un simple parcours de graphe permet de trouver une solution au jeu du Solitaire, par exemple. Les règles des jeux à plusieurs joueurs, comme le Tic-tac-toe, les échecs ou le Monopoly, peuvent aussi se définir comme des transitions entre états. Comme il y a plusieurs joueurs, et parfois du hasard, trouver une stratégie pour ces jeux demande des algorithmes plus complexes qu'un simple parcours de graphes, mais ces algorithmes reposent sur des idées similaires.

DEVINETTE Le chou, la chèvre et le loup

Une devinette raconte l'histoire d'un berger qui possède un chou, une chèvre et un loup. En sa présence, la chèvre n'ose pas manger le chou, pas plus que le loup n'ose manger la chèvre, mais ils n'hésiteraient pas à satisfaire leurs appétits si l'homme tournait le dos. Ce berger doit traverser une rivière avec sa petite troupe et il ne dispose que d'une barque, dans laquelle il peut naviguer avec un seul de ses compagnons. Comment doit-il s'y prendre ?

Ce problème peut lui aussi se modéliser comme un parcours de graphe. Tout d'abord, chacun des quatre protagonistes pouvant se trouver sur une rive ou l'autre, il y a seize états possibles :

	rive de départ	rive d'arrivée
0	–	chou, chèvre, loup, berger
1	chou	chèvre, loup, berger
2	chèvre	chou, loup, berger
3	chou, chèvre	loup, berger
4	loup	chou, chèvre, berger
5	chou, loup	chèvre, berger
6	chèvre, loup	chou, berger
7	chou, chèvre, loup	berger
8	berger	chou, chèvre, loup
9	chou, berger	chèvre, loup
10	chèvre, berger	chou, loup
11	chou, chèvre, berger	loup
12	loup, berger	chou, chèvre
13	chou, loup, berger	chèvre
14	chèvre, loup, berger	chou
15	chou, chèvre, loup, berger	–

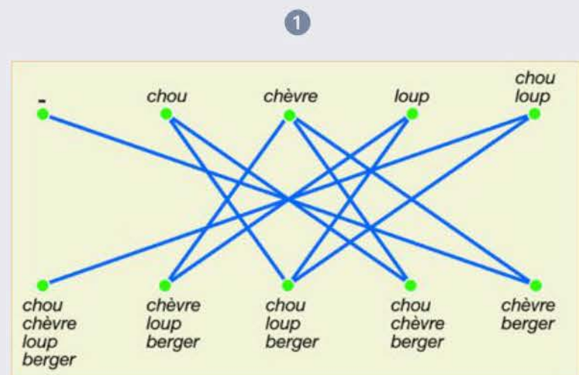
Les états 3, 6, 7, 8, 9 et 12 doivent être éliminés, car dans chacun d'eux le loup et la chèvre, ou la chèvre et le chou, sont sur une rive sans le berger. Il reste donc dix états que l'on peut nommer en fonction des protagonistes présents sur la rive de départ :

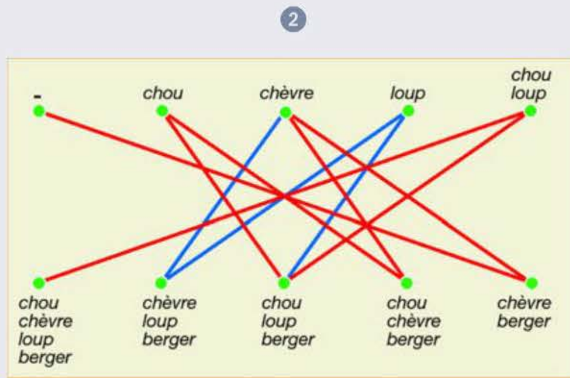
```
–
chou
chèvre
loup
chou, loup
chèvre, berger
chou, chèvre, berger
chou, loup, berger
chèvre, loup, berger
chou, chèvre, loup, berger
```

Définissons les transitions entre états : on ne peut pas passer directement de l'état *chou, chèvre, loup, berger* à l'état – car cela signifierait que le berger emporte à la fois le chou, la chèvre et le loup sur sa barque. En revanche, on peut passer de l'état *chèvre, loup, berger* à l'état *chèvre*, car cela signifie que le berger emmène le loup sur sa barque de la rive de départ à la rive d'arrivée. On peut de même passer de l'état *chèvre* à l'état *chèvre, loup, berger*, car cela signifie que le berger ramène le loup sur sa barque de la rive d'arrivée à la rive de départ. De manière générale, on peut passer d'un état qui contient le berger à un état dans lequel il n'est plus, ainsi qu'un ou zéro de ses compagnons, et vice-versa.

Sur ces bases, on définit le graphe (1) dont les sommets sont les dix états possibles et les arêtes les transitions entre ces états. Il ne reste plus qu'à utiliser un algorithme de parcours de graphe pour chercher un chemin qui mène de la configuration initiale *chou*, *chèvre*, *loup*, *berger*, à la configuration finale – (2).

1





Le chemin le plus court comporte sept arêtes, ce qui oblige le berger à faire quatre allers et trois retours :

- Il emmène d'abord la chèvre sur l'autre rive, ce qui correspond à l'arête qui va du sommet *chou*, *chèvre*, *loup*, *berger* au sommet *chou*, *loup*.
- Il revient seul.
- Il emmène le loup sur l'autre rive.
- Il revient avec la chèvre.
- Il emmène le chou.
- Il revient seul.
- Il emmène enfin la chèvre une seconde fois.

Bref, l'algorithme a « deviné » que la solution consiste à faire revenir la chèvre avec le berger pour ne jamais la laisser ni avec le loup ni avec le chou.

Exercice 22.1

Chercher sur le Web qui était Léonard Euler et décrire le problème des ponts de Königsberg, premier problème explicitement exprimé avec un graphe dans l'histoire des sciences.

Exercice 22.2

Décrire les objets suivants comme des graphes :

1. un réseau d'ordinateurs,
2. des maisons avec des boîtes aux lettres,
3. le métro parisien,
4. une ville, dans laquelle il n'y a plus ni radio ni télé, et dans laquelle se propage par SMS l'information de l'arrivée d'un tsunami,
5. une population au sein de laquelle se propage une épidémie.

Que sont les sommets ? Que sont les arêtes ? À quoi correspond la longueur du plus court chemin entre deux points ? Trouver d'autres exemples.

Exercice 22.3

Décrire un réseau social comme un graphe. Que sont les sommets ? Que sont les arêtes ?

Exercice 22.4

Imaginer un réseau social où ne sont en lien que les filles et les frères et sœurs : quel est le chemin minimal pour que deux garçons, qui ne sont pas de la même famille se passent un message à travers les arêtes ?

Exercice 22.5

Imaginer un autre réseau où chaque personne a dix amis. Combien de personnes au maximum un message diffusé aux amis des amis des amis, etc. peut-il atteindre en une heure, si le délai entre la réception et le rediffusion d'un message est de cinq minutes ? Que nous apprend ce résultat sur le risque de propagation d'une rumeur sur un réseau social ?

Exercice 22.6

Sur le plus grand réseau social du monde, il y a environ un milliard de personnes connectées chacune à cent autres personnes environ. Un chemin de deux arêtes – ami d'ami – connecte donc une personne à environ 100 × 100 = 10 000 autres personnes, si on néglige les amis communs et, disons, à environ 100 × 50 = 5 000 si l'on tient compte des amis communs. Un chemin de trois arêtes – ami d'ami d'ami – connecte une personne à environ 100 × 50 × 50 = 250 000 autres personnes. En supposant grossièrement qu'on ne gagne ainsi que la moitié de nouveaux contacts à chaque arête, à combien de personnes environ est-on connecté avec un chemin de six arêtes ? Dans un tel réseau social, quelle est la distance maximale entre deux personnes ? Il se trouve que ce calcul approximatif donne un résultat très proche de la réalité.

ALLER PLUS LOIN **Qu'est-ce qu'un algorithme fondamental ?**

Beaucoup de problèmes peuvent se formuler comme des problèmes de tri ; de même, on a vu dans ce chapitre que beaucoup de problèmes peuvent se ramener à des problèmes de graphes. Cela fait des algorithmes de tris et ceux de parcours de graphes des *algorithmes fondamentaux*. D'autres exemples d'algorithmes fondamentaux, que l'on peut rechercher sur le Web, sont ceux de multiplication de matrices, de résolution d'équations booléennes, d'unification... Quand on essaie de résoudre un problème, une bonne attitude consiste souvent à chercher à quel problème connu on peut se ramener et quel algorithme fondamental on peut utiliser.

Ai-je bien compris ?

- Que faut-il éviter quand on cherche la sortie d'un labyrinthe ?
- Quel est l'avantage de chercher la sortie d'un labyrinthe en largeur d'abord ?
- Quels objets peut-on décrire comme des graphes ?

TÉMOIGNAGE **Jonathan, 24 ans**

« J'ai commencé la programmation avec une (mauvaise) réplique de Mastermind qui permettait de jouer contre l'ordinateur. Après quelques essais manqués, quelques programmes ici et là, je découvre le Web : c'est le coup de foudre – je n'ai jamais décroché depuis. D'abord des sites web en PHP, puis le monde de Mozilla Firefox. De cette rencontre, naît un livre paru chez Eyrolles, rédigé pendant mon temps libre en terminale. Quelques années plus tard, je reprends l'informatique à l'École Normale Supérieure en section informatique : j'y découvre le lambda calcul et la programmation fonctionnelle. Je suis maintenant en thèse à Inria, dans l'équipe qui conçoit le langage OCaml, et je continue d'améliorer le code de Firefox et de Thunderbird sur mon temps libre au sein de la communauté Mozilla. »

TÉMOIGNAGE **Grégoire, 27 ans**

« Tombé dans la marmite de l'électronique et de la « bidouille », grâce à son papa, Grégoire passe rapidement à l'informatique et décide d'apprendre le C. S'ensuit une vraie passion pour les technologies dites nouvelles, et pour le lien intime et complexe entre matériel et logiciel. Math-sup, math-spé (pour faire plaisir à maman !), puis une formation d'ingénieur, et un rêve américain, concrétisé lors d'un stage dans la Silicon Valley à travailler sur une puce d'encodage vidéo. Grégoire revient ensuite en France et, à Paris, met sa créativité à profit pour inventer la « set-top box » de demain, qui tient au creux de la main et propose une expérience visuelle digne des derniers jeux vidéos. À titre personnel, il songe à des manières de simplifier l'informatique et la programmation, mais sans perdre le lien avec le matériel, et développe secrètement le concept de « Software Atoms ».

Convaincu que la valeur et l'innovation sont portés par ceux qui créent (les programmeurs et les passionnés), il milite pour le développement de hiérarchies d'entreprise moins verticales, pour la revalorisation du développement logiciel en France et de la créativité technologique. »

Idées de projets

Un générateur d'exercices de calcul mental

Programmer un générateur d'exercices de calcul mental : le programme choisit aléatoirement une opération et deux nombres, et vérifie la réponse de l'utilisateur. On peut ensuite poser une série de questions et compter le score total. On pourra enfin prévoir plusieurs niveaux de difficulté selon les opérations proposées ou la taille des nombres à calculer, et laisser l'utilisateur choisir son niveau de difficulté ou attribuer des scores variables aux réponses.

Mastermind

Écrire un programme qui lit deux tableaux de quatre éléments au clavier et indique le nombre d'éléments en commun dans ces deux tableaux. Écrire un programme qui joue au Mastermind : tire au hasard la combinaison secrète et répond aux propositions de l'autre joueur.

Brin d'ARN

Chercher sur le Web ce qu'est un brin d'ARN messager et comment il code pour une protéine. Écrire un programme qui détermine la protéine pour laquelle un brin d'ARN messager code.

Bataille navale

Écrire un programme de bataille navale avec plusieurs bateaux.

Cent mille milliards de poèmes

Chercher sur le Web, ou dans une bibliothèque, ce qu'est le recueil *Cent mille milliards de poèmes* de Raymond Queneau. Écrire un programme qui affiche ces *cent mille milliards de poèmes*.

Site de rencontres

Programmer le moteur d'un site de rencontres, sur le principe suivant :

- Chaque personne inscrite sur le site répond à un questionnaire de personnalité dont les réponses sont des entiers entre 1 et 10.
- On stocke les réponses dans un tableau à double entrée : la case de la ligne `i` et de la colonne `j` contient la réponse de l'inscrit numéro `i` à la question numéro `j`.
- Pour mettre en relation un nouvel inscrit avec une personne déjà inscrite, on parcourt ce tableau en recherchant la ligne qui contient les réponses les plus proches de celles données par le nouvel inscrit. Pour déterminer si des réponses sont proches, on pourra par exemple compter le nombre de réponses identiques ou calculer le total des différences.
- On pourra enfin, à partir d'un tableau déjà rempli, chercher à former autant de couples que possible.

Témoignage Christine

« Quand j'étais adolescente, il n'y avait d'ordinateurs ni à l'école, ni à la maison. Pourtant nos cours comportaient de nombreuses activités liées à l'informatique : représenter la même information dans plusieurs formats, élaborer différentes méthodes pour réaliser le même objectif (je me souviens de cartes perforées et d'aiguilles à tricoter pour trier) ou encore raisonner et calculer à partir d'un ensemble donné de règles.

Ma première calculatrice ? Je ne l'ai eue qu'après le baccalauréat et, si elle était programmable, c'était à partir d'un jeu d'instructions très élémentaire. Cela n'en était qu'un défi plus intéressant. Face à une série de problèmes à résoudre, élaborer la bonne séquence d'instructions qui permettra d'obtenir toutes les solutions sans effort est gratifiant, et aussi distrayant que de résoudre un casse-tête. Contrairement à un jeu tout fait, le champ des possibles est immense : les moyens de calcul, de communication, et les données, sont à notre portée pour pouvoir les explorer.

Écrire soi-même un programme qui marche requiert connaissances, expérience, et imagination. La satisfaction est immense d'avoir créé un nouvel objet.

Ce plaisir que j'ai eu à transformer les problèmes en programmes est encore intact après 25 ans de carrière de chercheuse au CNRS puis de professeur à l'université. J'ai choisi l'informatique comme métier (assez tardivement) grâce à un professeur qui m'a fait découvrir tout ce que cette discipline nouvelle offrait d'opportunités en termes de métier et d'activité. J'y ai trouvé une voie qui répondait à mes goûts et compétences. Aujourd'hui, j'essaie de donner aux étudiantes et étudiants les clés théoriques et pratiques du monde numérique qu'ils « consomment » chaque jour, pour qu'ils sachent l'adapter à leurs besoins et contribuent à l'enrichir. »

Tracer la courbe représentative d'une fonction polynôme du second degré

Écrire un programme qui, étant donné une fonction polynôme du second degré, trace sa courbe représentative à l'écran en adaptant automatiquement la fenêtre pour faire apparaître le sommet de la parabole et les éventuels zéros. On pourra faire réaliser les calculs intermédiaires dans des fonctions séparées.

Gérer le score au tennis

Écrire un programme qui gère automatiquement le score au tennis :

1. À quelles conditions un joueur gagne-t-il un jeu ?
2. Définir une fonction qui compte les points au cours d'un jeu. En entrée, on demande répétitivement quel joueur, 1 ou 2, gagne le point ; au fur et à mesure, on calcule et on affiche le score. Le programme s'arrête dès qu'un joueur gagne le jeu, après avoir affiché le nom du vainqueur.
3. Pour faciliter les tests, écrire une seconde version de cette fonction avec pour seule entrée une chaîne de caractères qui contient les numéros successifs des joueurs marquant les points ; la fonction lit cette chaîne caractère par caractère pour compter les points. Ainsi, l'entrée 211222 sera comprise comme : le joueur 2 gagne un point, puis le joueur 1 en gagne deux, puis le joueur 2 en gagne trois et le joueur 2 gagne donc le jeu.
4. Écrire une deuxième fonction qui compte les jeux au cours d'un set et s'arrête lorsqu'un joueur gagne le set. Cette fonction fera appel à la précédente pour savoir qui gagne les jeux. On n'oubliera pas de prévoir le cas particulier du jeu décisif.
5. Écrire une troisième fonction qui compte les sets et s'arrête lorsqu'un joueur gagne le match. On pourra, avant de commencer le match, demander en combien de sets gagnants il est joué.

Automatiser les calculs de chimie

Programmer une boîte à outils pour automatiser les différents calculs que l'on a l'occasion de faire en chimie : durée d'une réaction, pH d'une solution, masses et concentrations, équilibre d'une équation-bilan simple…

Tours de Hanoï

Chercher sur le Web ce que sont les tours de Hanoï et écrire un programme qui trouve une solution à ce jeu.

Tortue Logo

Chercher sur le Web ce qu'est une tortue Logo. Programmez ses principales fonctionnalités. Chercher sur le Web ce qu'est le flocon de Von Koch et dessiner ce flocon à l'aide de cette tortue.

Dessins de plantes

Programmer des dessins de plantes suivant un modèle récursif, par exemple une fougère. On pourra introduire un élément aléatoire dans l'algorithme, afin de varier les résultats obtenus. On pourra utiliser une tortue Logo programmée par soi-même ou par un camarade, ou fournie dans une bibliothèque du langage de programmation utilisé.

Langage CSS

Rechercher sur le Web comment le langage CSS permet de donner une présentation différente à des informations, selon qu'elles sont lues sur un ordinateur ou sur l'écran d'un téléphone. Construire un site Web sur le sujet de son choix qui peut ainsi être consulté sur un écran ou un autre.

Calcul sur des entiers de taille arbitraire

En représentant chaque nombre par un tableau qui contient la suite de ses chiffres, écrire une bibliothèque de fonctions calculant sur des entiers de taille arbitraire, sans les dépassements de capacité qu'engendre l'utilisation du type `int`.

TÉMOIGNAGE **Raphaël, 35 ans**

« L'informatique, je suis tombé dedans très jeune, et mon premier programme en BASIC, qui affichait une carte de France, m'a fait découvrir le plaisir de dominer l'ordinateur. Avec un peu d'attention, on peut lui faire faire ce que l'on veut. Après avoir découvert Windows, je suis devenu un grand fan de Bill Gates. Mais alors que j'étais au lycée, Internet est arrivé et grâce à lui, j'ai redécouvert l'informatique sous un nouvel angle. Je suis entré dans un nouvel univers, celui du logiciel libre. Finis les bricolages : il était désormais possible de diagnostiquer tous les problèmes, voire de les corriger grâce à l'accès au code source. Lorsque cela dépassait mes compétences, je pouvais contacter l'auteur du logiciel et obtenir un correctif en l'espace de quelques jours.

Très rapidement, j'ai ressenti le besoin de m'impliquer dans cette communauté pour apporter ma pierre à l'édifice. C'est ainsi que je suis devenu développeur Debian, distribution GNU/Linux, alors que j'entrais à l'INSA de Lyon. Deux ans après avoir obtenu mon diplôme d'ingénieur en informatique, j'écrivais le premier livre français sur Debian et je lançais ma propre société, Freexian, pour faire de ma passion, mon activité principale. »

Calcul en valeur exacte sur des fractions

Proposer un type représentant une fraction en valeur exacte. Écrire une bibliothèque de fonctions pour effectuer des calculs en valeur exacte sur des fractions. Rechercher sur le Web ce qu'est la méthode de Héron pour calculer une valeur approchée d'une racine carrée et la programmer en utilisant la bibliothèque de calcul sur les fractions.

Représentation des dates et heures

Les systèmes numériques passent automatiquement à l'heure d'été et détectent quand on change de fuseau horaire. Chercher sur le Web des informations sur la représentation des dates et heures : la norme ISO 8601 et le *Network Time Protocol*. Écrire un programme qui donne l'heure dans différents pays.

Transcrire dans l'alphabet latin

Choisir une langue qui utilise un autre alphabet que l'alphabet latin (par exemple le chinois, l'arabe, le japonais, l'hébreu ou le grec) et une trentaine de mots écrits dans cette langue. Associer à chaque syllabe de l'un de ces mots une transcription phonétique écrite dans l'alphabet latin. Écrire un programme d'apprentissage qui tire au hasard des mots dans cette liste, les affiche à l'écran et demande à l'utilisateur de les lire, c'est-à-dire de les transcrire dans l'alphabet latin.

Correcteur orthographique

Réaliser un correcteur orthographique. Un tel programme prend en entrée un fichier texte, le découpe en mots, cherche chaque mot dans un dictionnaire et donne la liste des mots qui ne s'y trouvent pas.

Daltonisme

Rechercher sur le Web ce qu'est le daltonisme et quelles en sont les différentes formes. Écrire un programme qui lit une image dans un fichier au format PPM et l'affiche à l'écran comme la verrait une personne atteinte de chacune des formes de daltonisme.

Système audio par syllabe

Enregistrer un fichier audio par syllabe « ba », « be », « bi », « bo », « bu », « bou », « bon », etc. Utiliser ces fichiers pour écrire un programme qui envoie une séquence de ces grains sonores au système audio, en fonction d'un texte écrit phonétiquement « bon », « jou », « re », « i », « lé », « ne », « ve », « re ». Comparer le résultat avec un système professionnel de synthèse vocale et mettre en lumière les difficultés d'un tel mécanisme.

Déchiffrer automatiquement un message codé selon la méthode de César

Écrire un programme qui déchiffre automatiquement un message codé selon la méthode de César sans connaître a priori la valeur du décalage. Rechercher ce qu'est le chiffre de Vigenère et écrire un programme qui code un message selon ce principe. Rechercher une méthode pour décoder un message codé selon ce principe sans connaître la clé utilisée, et écrire un programme qui effectue ce décodage.

Logisim

Pour construire et simuler des circuits, on peut utiliser logiciel Logisim disponible à l'adresse :

http://ozark.hendrix.edu/~burch/logisim/

Ce logiciel libre fonctionne sur la plupart des ordinateurs. Il n'est pas encore traduit en français, mais il est très bien documenté : on peut commencer sa lecture par le tutoriel. En utilisant ce logiciel, on peut par exemple construire le multiplexeur et les circuits de décalage définis au chapitre 13 et le compteur huit bits défini au chapitre 14.

Banc de registres

Rechercher sur le Web ce qu'est un banc de registres, ainsi que les notions de port de lecture et de port d'écriture sur un banc de registres. À l'aide de l'horloge et des circuits vus aux chapitres 13 et 14, réaliser un banc de 8 registres 8 bits et dessiner le circuit correspondant.

Témoignage Pierre, 24 ans

« J'étais en sixième, lorsque mon premier ordinateur est arrivé à la maison, mais je dois reconnaître que pendant bien longtemps, il n'avait d'utilité pour moi que comme simple console de jeux. C'est en classes préparatoires, sur les conseils de mon professeur de mathématiques de l'époque, que j'ai choisi de prendre l'option Informatique plutôt que Sciences Industrielles. Je n'avais aucune idée encore de ce que pouvait être un langage de programmation ni même de la manière dont fonctionnait un ordinateur en général. Se lancer dans une matière totalement inconnue alors qu'on a un emploi du temps déjà chargé n'est pas un choix aisé. Il s'est cependant révélé judicieux : j'étais alors porté sur les mathématiques mais j'ai retrouvé dans l'enseignement de l'informatique certains concepts (ensembles, fonctions…), tout en évitant le calcul (et les erreurs qui vont avec) que je détestais. C'est en école d'ingénieur que mon choix de spécialisation s'est définitivement porté sur l'informatique. Au fil de mes lectures diverses je me faisais (enfin) une idée de ce que je voudrais faire « plus tard » : de la logique, comprendre comment sont construits le raisonnement et les objets des mathématiques. J'ai donc commencé ma quête dans le département de mathématiques où l'on m'a expliqué que ces aspect étaient plutôt étudiés dans le laboratoire d'à côté… Là, j'ai retrouvé un professeur qui m'avait impressionné l'année précédente en présentant les modifications des cases mémoires d'un ordinateur, à l'aide de boites à chaussures. Après un master de recherche en informatique, j'ai commencé ma thèse dans le cadre d'un partenariat entre Inria et la NASA. Je développe un algorithme pour améliorer la précision des calculs sur les ordinateurs. »

Simuler le comportement d'un processeur

Écrire un programme simulant le comportement du processeur lorsqu'il doit exécuter un des programmes écrits en langage machine décrits au chapitre 15. Ce programme lira dans un fichier le contenu de la mémoire, qui contient également le programme à exécuter. Il affichera l'état de la mémoire et des registres au fur et à mesure de l'exécution.

Effectuer des calculs sur les adresses de cases mémoires

Le langage machine décrit au chapitre 15 ne permet d'accéder qu'à un ensemble fini de cases mémoires, dont les adresses sont fournies dans le code des instructions LDA, STA, etc. Pour effectuer des calculs complexes, et notamment pour manipuler des tableaux, on doit pouvoir effectuer des calculs sur les adresses de cases mémoires elles-mêmes. Proposer une extension du langage machine pour effectuer de tels calculs, en définissant la syntaxe des nouvelles instructions, en choisissant leur code machine (binaire) et en expliquant leur fonctionnement. Programmer des boucles simples réalisant des calculs sur des tableaux, par exemple : la somme des éléments d'un tableau, le compteur du nombre d'éléments positifs, etc.

Utilisation du logiciel Wireshark

Installer et lancer le logiciel Wireshark. Capturer des paquets Ethernet ou WiFi depuis la carte réseau et afficher leur contenu à l'écran. Quelles sont les adresses MAC utilisées pour la destination et la source de chaque paquet ? Quels ordinateurs sont identifiés par ces adresses ? Quelle est la taille de chaque paquet ?

Algorithme de pledge

Chercher sur le site web interstices.info ce qu'est l'algorithme de *pledge*. Programmer cet algorithme. Expliquer son utilité et en quoi il se distingue de l'algorithme de sortie d'un labyrinthe du chapitre 22.

Algorithme calculant le successeur d'un nombre entier naturel *n*

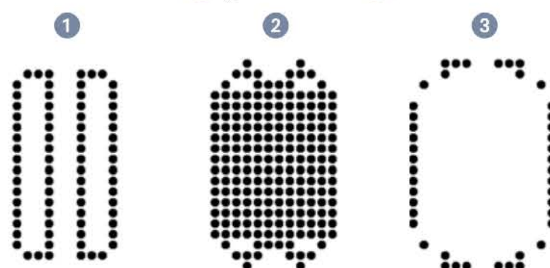
On considère un algorithme qui calcule le successeur d'un nombre entier naturel n, c'est-à-dire le nombre $n + 1$. Cet algorithme est similaire à celui de l'addition, mais il s'applique à un unique nombre : il procède de la droite vers la gauche en posant un chiffre et en propageant une retenue à chaque étape. Identifier les fonctions booléennes qui à un chiffre binaire et une retenue associent le chiffre à poser et la retenue à propager. Programmer cet algorithme et démontrer sa correction en suivant les lignes de la démonstration de correction de l'algorithme de l'addition (voir le chapitre 18). Pour aller plus loin : dessiner un circuit booléen (voir le chapitre 13) qui ajoute 1 à un nombre binaire de quatre bits.

Le jeu de la vie

Sur un damier carré, on dispose des créatures de manière aléatoire. La population évolue d'un état au suivant selon les règles suivantes :

- Une créature survit si elle a 2 ou 3 voisines dans les 8 cases adjacentes et elle meurt à cause de son isolement ou de la surpopulation sinon.
- Une créature naît dans une case vide s'il y a exactement 3 créatures dans les 8 cases voisines, et rien ne se passe dans cette case sinon.

Par exemple, en partant de l'état initial ❶, la population évolue à l'état ❷, puis à l'état ❸, etc.



Écrire un programme qui simule le développement d'une population.

TÉMOIGNAGE **Dominique, 36 ans**

« Je suis artisan de profession ; mon métier consiste à programmer des ordinateurs.

J'avais 9 ans lorsque mon père, professeur de mathématiques, rapporta une calculatrice TI-57 à la maison. Je me souviens encore de mon tout premier programme, expliqué dans le manuel avec des organigrammes dessinés sous la forme de petites voies ferrées que la locomotive-microprocesseur parcourt : [LRN] ("Learn" - Passer en mode programmation), [+], [1], [=], [RST] (revenir au début), [LRN], [RST], [R/S] (Run / Stop). Et c'est parti, la calculatrice compte 1, 2, 3, 4… L'exercice suivant — programmer le jeu de « devine un nombre » — me vit mettre en œuvre toute mon habileté et mon astuce pour tenir dans les 40 pas de programme disponibles dans l'appareil. J'étais pris au jeu ! Et de MO-5 en Atari ST, et puis de Logo en Turbo Pascal, j'ai appris à programmer. Le temps a passé, et matériels et logiciels ont progressé à pas de géant.

Vous qui apprenez l'informatique aujourd'hui, vous vous moquerez peut-être gentiment de la TI-57. Vous auriez tort, car programmer le jeu de « devine un nombre » dans le langage de votre choix est le genre de question qu'on pourrait vous poser lors d'un entretien téléphonique pour une embauche chez Google (où je travaille depuis fin 2007). Les admirables fondements mathématiques de la discipline nous enseignent que les ordinateurs se ressemblent tous ; mais pas les humains qui s'en servent ou qui les programment. Ces derniers se distinguent des premiers parce qu'avec l'ignorance disparaît la peur de l'instrument qui révolutionne nos vies ; et les programmeurs se distinguent entre eux par leurs centres d'intérêt parmi les nombreuses spécialités de l'informatique, leurs langages et styles de programmation préférés, et le but qu'ils poursuivent à titre personnel ou professionnel. Mais tous les programmeurs, ou presque, partageons le goût d'apprendre, l'attention au travail bien fait et l'esprit de géométrie, cher à Blaise Pascal.

Que vous souhaitiez ou non en faire votre métier, je ne saurais trop vous recommander d'aborder la programmation comme j'ai abordé la guitare (et non le violon) : en commençant par vous amuser, puis en continuant par vous perfectionner, comme un forgeron qui cent fois sur son enclume remet son ouvrage. La perfection existe en informatique ; c'est même une joie inépuisable que de se mettre à sa quête, en parcourant un territoire immense et encore si largement inexploré ! »

Une balle

Dessiner une balle qui rebondit sur les parois de la fenêtre graphique. Écrire pour cela une fonction qui dessine une balle sphérique à un endroit donné de l'image, une autre fonction qui calcule si cette balle touche le bord de la fenêtre ou non, une troisième qui calcule sa position suivante selon qu'elle rebondit sur le bord ou non. On pourra prendre en compte la gravité, le ralentissement de la balle à cause des frottements, etc. Produire une suite de quelques centaines d'images et agglomérer cette suite sous la forme d'un film en utilisant un logiciel de création de vidéos ou de GIF animés.

Générateur d'œuvres aléatoires

Trouver des tableaux sur le site web d'un musée. Choisir aléatoirement un petit détail d'un de ces tableaux et agrandir ce détail en un nouveau tableau. Changer les couleurs, le contraste, mélanger des détails issus de deux tableaux différents, etc. Vérifier la licence de ces images et comprendre s'il est possible de publier ses résultats ou non.

Détecteur de mouvement visuel

À l'aide d'une webcam, prendre deux photos successives avec un délai minimal entre les deux, soustraire pixel à pixel ces deux photos et stocker l'image obtenue. Écrire un programme qui utilise un seuil pour détecter dans cette image un mouvement non négligeable et qui donne la taille en pixels de la tache de mouvement obtenue. Tester ce procédé en situation réelle. Quelles en sont les possibilités et les limites ? Appliquer ce procédé à un objet qui « donne un coup de boule » à la caméra, c'est-à-dire qui s'en approche à vitesse constante le long de son axe optique. Avec un modèle géométrique très simple, où l'on considère l'objet comme un rectangle plat parallèle à la caméra, calculer le temps restant avant la collision avec la caméra. Vérifier expérimentalement les résultats obtenus.

Qui est-ce ?

Créer une version numérique et graphique du jeu de société « Qui est-ce ? ». Quelle est l'utilité de la notion de dichotomie pour jouer à ce jeu ?



Un joueur de Tic-tac-toe

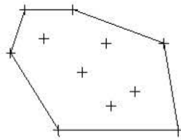
Le Tic-tac-toe est un jeu où deux joueurs placent à tour de rôle l'un des ronds O et l'autre des croix X sur un plateau de trois cases sur trois cases, jusqu'à ce que l'un des joueurs ait aligné trois symboles ou que les neuf cases soient remplies. C'est le joueur O qui commence. Décrire ce jeu sous forme d'un ensemble d'états et d'un ensemble de transitions. Combien y a-t-il d'états possibles ? Un état du jeu peut être gagnant pour O, gagnant pour X, match nul ou en cours de jeu. Exprimer chacun des 3^9 états à l'aide d'un nombre entier. Construire un tableau qui, pour chacun de ces états, indique s'il est gagnant pour l'un des joueurs, nul ou en cours de jeu et, dans ce cas, les transitions possibles pour chacun des joueurs. Calculer ensuite, de proche en proche, les états dans lesquels :

- le joueur O est certain de gagner,
- le joueur X est certain de gagner,
- le joueur O est certain de gagner ou de faire match nul,
- le joueur X est certain de gagner ou de faire match nul.

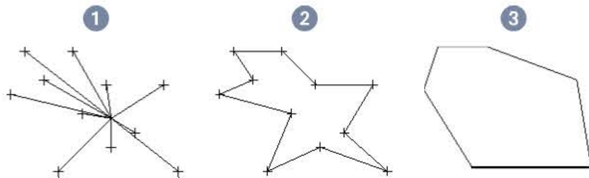
Écrire un programme qui joue contre un joueur humain.

Enveloppe convexe

Écrire un programme qui dessine l'enveloppe convexe d'un ensemble fini de points du plan.



Une manière de faire consiste à trier ces points par angle polaire croissant (1), à relier entre eux les points consécutifs (2), puis à supprimer l'un après l'autre les angles rentrants (3).



Chemins les plus courts

L'algorithme de parcours d'un graphe en largeur d'abord permet de déterminer s'il existe un chemin entre deux sommets d'un graphe et de calculer un plus court chemin, si ces deux sommets sont effectivement reliés. L'algorithme de Roy-Warshall-Floyd va plus loin en déterminant une fois pour toutes s'il existe un chemin entre toutes les paires de sommets d'un graphe et en calculant un plus court chemin pour chaque paire de sommets effectivement reliés.

Plus précisément, cet algorithme calcule deux tableaux :

- un tableau `L` tel que `L[x][y]` soit la longueur du plus court chemin reliant le sommet `x` au sommet `y` si ces deux sommets sont effectivement reliés et ∞ sinon,
- un tableau `R` tel que `R[x][y]` soit le premier sommet après `x` sur un plus court chemin reliant `x` à `y`, si ces deux sommets sont effectivement reliés, et si le plus court chemin les reliant n'est pas de longueur nulle, c'est-à-dire si `x` est différent de `y`.

Comme dans un jeu de pistes, on peut retrouver l'intégralité de ce chemin en partant de `x`, en allant en `R[x,y]`, puis en `R[R[x,y],y]`, etc. jusqu'à arriver en `y`.

Pour calculer ces deux tableaux, on commence par initialiser le tableau `L` en mettant dans la case `L[x][y]` la valeur 0 si `x` est égal à `y`, la valeur 1 si `x` est différent de `y` et s'il y a une arête qui relie `x` à `y`, et la valeur ∞ sinon. On initialise de même le tableau `R` en mettant dans la case `R[x][y]` la valeur `y`.

Ensuite, on imbrique trois boucles :

- pour tous les sommets intermédiaires `z`,
- pour tous les sommets de départ `x`,
- pour tous les sommets d'arrivée `y`,

si `L[x][z] + L[z][y] < L[x][y]`, c'est-à-dire si aller de `x` à `y` en passant par `z` est strictement plus court que le plus court chemin connu, on remplace `L[x][y]` par `L[x][z] + L[z][y]` et `R[x][y]` par `R[x][z]`.

Il n'est pas difficile de montrer que, après avoir effectué les tours de la boucle la plus externe correspondant aux sommets $z_1, \dots, z_k$, la case `L[x][y]` du tableau `L` contient la longueur du plus court chemin qui relie `x` à `y` en passant possiblement par les sommets $z_1, \dots, z_k$, mais pas par les autres sommets du graphe et que la case `R[x][y]` du tableau `R` contient le premier sommet de ce chemin, si un tel chemin existe et s'il n'est pas de longueur nulle.

Programmer cet algorithme et l'appliquer à un réseau de métro ou de bus.

Utilisation des réseaux sociaux

En utilisant le réseau social le plus répandu dans sa classe, et avec l'accord des personnes concernées, construire le graphe qui a pour sommets les élèves de sa classe et pour arêtes la relation « est l'ami de ». Au fur et à mesure de l'analyse, on pourra raffiner ses données en considérant des sommets collectifs, par exemple la participation à un club. Entrer ces données dans un tableau bidimensionnel dont chaque ligne et colonne correspond à un sommet et chaque case à une arête et calculer les composantes connexes de ce graphe, c'est-à-dire les sous-ensembles maximaux de sommets connectés. Déterminer le nombre de composantes connexes. Ce projet doit être réalisé en respectant l'anonymat des personnes.

Index